Geekbrains

**Разработка веб-приложения для управления складом и ведения заметок на базе Java Spring Boot**

Программа: Разработчик

Специализация: Программист

Адвахов Е.Л.

Кахул

2024

**Содержание**

**1. Введение**………………………………………..…………………………..……..1

1.1 Актуальность проекта………………………………………………………..1

1.2 Цели и задачи проекта……………………………………………………….1

1.3 Инструменты разработки…………………………………………………….3

**2. Проектная стадия**………………………………………………………………..5

2.1 Выбор шаблона проектирования ………………….. ………………………5

* 1. Проектирование архитектуры базы данных……………...………………...5
  2. Разработка требований к безопасности…………………………………….7
  3. Разработка стратегии тестирования………………………………………...8

**3. Практическая часть**…………………………………………………………....10

3.1 Создание классов – сущностей……………………………………………..10

3.2 Создание классов – сервисов……………………………………………….14

3.3 Создание классов – репозиториев………………………………………….16

3.4 Создание классов – котроллеров …………………………………………...16

3.5 Логирование …………………………………………………………………25

3.6 Подключение внешнего API ZenQuotes.io…………………………………27

3.7Разворачивание Docker контейнера и подключение базы данных MySQL………………………………………………………………………………29

3.8 Подключение конфигурационного процессора……………………………30

3.9 Безопасность ...………………………………………………………………31

3.10 Тестирование………………………………………………………………..34

3.11 Подключение системы мониторинга……………………………………...37

3.12 Реализация дизайна и функционала веб-приложения…………………...41

Заключение………………………………………………………………………….51

Список используемой литературы………………………………………………...52

**1. Введение**

**1.1 Актуальность проекта**

В современном мире организация складского учета является одной из ключевых задач для успешного функционирования любого предприятия. Однако часто встречаются случаи недостаточно хорошо организованной структуры учета, что приводит к многочисленным проблемам, таким как потеря или некорректное хранение данных о товарных запасах, затрудненный доступ к информации и отсутствие централизованного управления данными.

В процессе своей профессиональной деятельности я неоднократно сталкивался с такими недостатками в организации складского учета. Эти проблемы стали основным стимулом для разработки моего проекта – создания веб-приложения для учета склада. Данное решение позволит централизованно хранить всю необходимую информацию, обеспечивая удобный и быстрый доступ к данным в любое время.

Кроме того, веб-приложение будет обладать функционалом для ведения записей важных заметок, что значительно расширит его возможности и сделает его полезным не только для складского учета, но и для других аспектов управления предприятием.

Таким образом, актуальность данного проекта заключается в создании удобного и эффективного инструмента для улучшения организации складского учета и централизованного хранения информации, что позволит оптимизировать процессы управления и повысить общую эффективность работы предприятия.

**1.2** **Цели и задачи проекта**

Целью данного проекта является создание веб-приложения для эффективного учета складских операций и ведения важных заметок. Для достижения этой цели необходимо выполнить следующие задачи и учесть ключевые требования к приложению:

1. **Функциональные требования:**
   * **Права пользователя (User):** Пользователь с правами user должен иметь возможность заносить товар на склад, удалять его и редактировать. Это обеспечит гибкость и простоту управления товарными запасами.
   * **Права администратора (Administrator):** Пользователь с правами admin должен иметь тот же функционал, что и пользователь, плюс возможность создавать и удалять пользователей. Это позволит централизованно управлять доступом и правами других пользователей.
   * **Логирование действий:** Приложение должно вести журнал всех действий пользователей на складе. Это важно для обеспечения прозрачности и возможности аудита.
2. **Безопасность:**
   * **Хранение паролей:** Пароли пользователей должны храниться в зашифрованном виде, чтобы защитить их от несанкционированного доступа.
   * **Аутентификация и авторизация:** Необходимо реализовать надежные механизмы аутентификации и авторизации пользователей для предотвращения несанкционированного доступа к данным.
3. **Производительность:**
   * **Быстрый доступ к данным:** Приложение должно обеспечивать быстрый доступ к данным, чтобы пользователи могли оперативно получать необходимую информацию.
   * **Отзывчивый интерфейс:** Интерфейс приложения должен быть интуитивно понятным и быстро реагировать на действия пользователя, обеспечивая удобство и комфорт в работе. Простота использования позволит всем категориям пользователей легко освоить функционал приложения и эффективно с ним взаимодействовать.
4. **Масштабируемость:**
   * **Расширяемость функционала:** Приложение должно быть спроектировано с возможностью добавления новых функций в будущем, чтобы удовлетворять растущие потребности пользователей.
   * **Поддержка роста числа пользователей:** Приложение должно быть способно поддерживать рост числа пользователей без ухудшения производительности.

Эти цели и задачи позволят создать надежное и эффективное веб-приложение для учета складских операций, соответствующее современным требованиям безопасности, производительности и масштабируемости.

**1.3 Инструменты разработки**

Для реализации проекта были выбраны следующие инструменты и технологии:

**1.** **Язык программирования Java**: Основной язык программирования для разработки серверной части приложения. Java выбран за его широкие возможности, безопасность и масштабируемость, что делает его идеальным для создания веб-приложений.

**2.** **Фреймворк Spring Boot**: Используется для создания микросервисов и разработки RESTful API. Spring Boot позволяет быстро настроить и запустить приложение, обеспечивая гибкость и удобство в управлении зависимостями.

**3.** **Среда разработки IntelliJ IDEA**: Эта IDE предлагает мощные инструменты, которые значительно повышают продуктивность и удобство разработки.

**4.** **Система управления базой данных MySQL**: Используется для хранения данных приложения. MySQL – это надежная и производительная СУБД, которая обеспечивает удобство работы с большими объемами данных.

**5. Фронтенд:**

HTML: Используется для создания структуры веб-страниц. HTML является основным языком разметки для веб-разработки, обеспечивая базовую структуру интерфейса.

Bootstrap: Фреймворк для упрощения разработки интерфейса. Bootstrap предоставляет готовые компоненты и стили, что ускоряет процесс создания адаптивных и эстетичных веб-страниц.

Thymeleaf: Шаблонизатор, используемый для динамического рендеринга HTML-страниц. Thymeleaf интегрируется с Spring Boot и позволяет легко связывать данные из backend с frontend.

1. **Система управления версиями GitHub:** Платформа для управления версиями и хранения программного кода. GitHub позволяет эффективно вести разработку в команде, отслеживать изменения в коде и управлять различными версиями проекта.

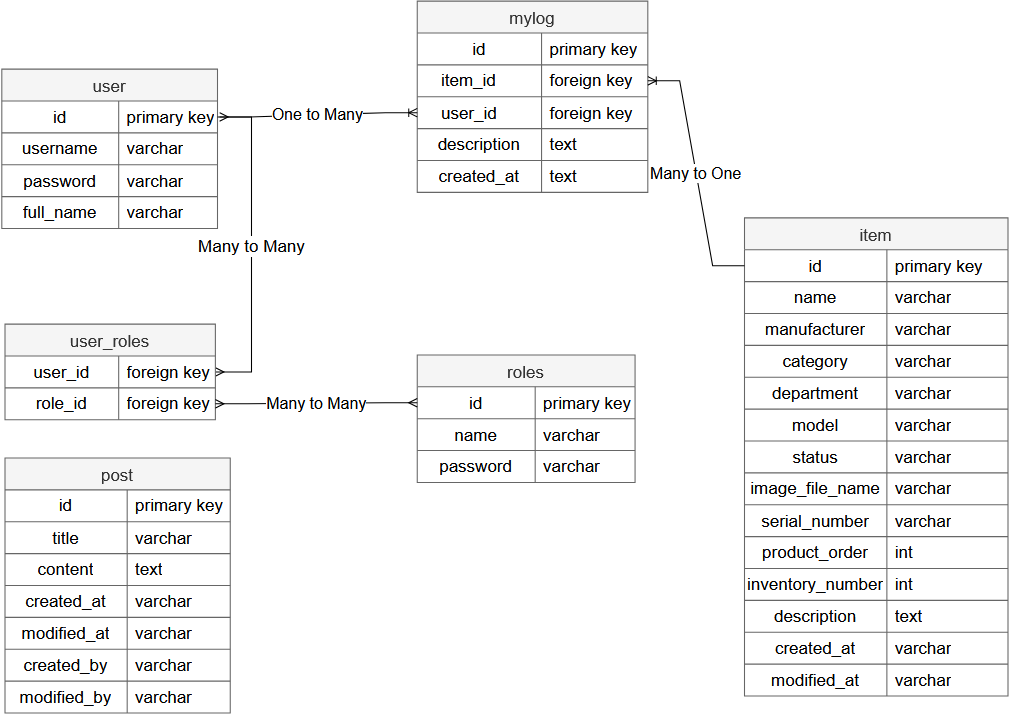
**2. Проектная стадия**

**2.1 Выбор шаблона проектирования**

Для разработки веб-приложения был выбран шаблон проектирования Model-View-Controller (MVC). Этот шаблон обеспечивает четкое разделение логики приложения на три взаимосвязанных компонента:

* **Model (Модель):** Отвечает за управление данными и бизнес-логикой приложения.
* **View (Представление):** Отображает данные и отвечает за пользовательский интерфейс.
* **Controller (Контроллер):** Обрабатывает пользовательские запросы, взаимодействует с моделью и обновляет представление.

**2.2 Проектирование архитектуры базы данных**

****

**Взаимоотношения сущностей:**

1. **User (Пользователь):**
   * Связь "One to Many" с сущностью **mylog** (Журнал): Один пользователь может иметь много записей в журнале.
   * Связь "Many to Many" с сущностью **roles** (Роли) через таблицу **user\_roles**: Один пользователь может иметь много ролей, и одна роль может принадлежать многим пользователям.
2. **Mylog (Журнал):**
   * Связь "Many to One" с сущностью **user**: Многие записи в журнале могут быть связаны с одним пользователем.
   * Связь "Many to One" с сущностью **item** (Товар): Многие записи в журнале могут быть связаны с одним товаром.
3. **Item (Товар):**
   * Связь "One to Many" с сущностью **mylog**: Один товар может иметь много записей в журнале.
4. **Roles (Роли):**
   * Связь "Many to Many" с сущностью **user** через таблицу **user\_roles**: Одна роль может принадлежать многим пользователям, и один пользователь может иметь много ролей.
5. **User\_roles (Пользовательские роли):**
   * Таблица связывает пользователей и их роли в отношении "Many to Many".
6. **Post (Пост):**
   * Эта сущность не связана с другими сущностями на диаграмме. Она содержит информацию о постах, таких как заголовок, содержание, даты создания и изменения, а также автор поста.

**Пример объяснения на конкретных случаях:**

* **Пользователь и Роли:** Один пользователь может обладать несколькими ролями. Например, пользователь может быть как ***admin***, так и просто ***user*** склада. Это достигается через промежуточную таблицу **user\_roles**, которая связывает пользователя и его роли.
* **Пользователь и Журнал:** Записи в журнале отражают действия, выполненные пользователем, такие как добавление или удаление товаров. Один пользователь может иметь множество записей в журнале, что показывает историю его действий.
* **Товары и Журнал:** Записи в журнале также связаны с товарами. Например, одна запись может описывать, что определенный товар был добавлен на склад, а другая запись может описывать его редактирование.

**Дополнительные детали:**

* **User (Пользователь):** Хранит информацию о пользователях системы, включая имя пользователя, пароль и полное имя.
* **Roles (Роли):** Хранит информацию о ролях, которые могут быть присвоены пользователям.
* **Mylog (Журнал):** Хранит историю действий пользователей по отношению к товарам.
* **Item (Товар):** Хранит информацию о товарах на складе, включая имя, производителя, категорию и другие атрибуты.
* **Post (Пост):** Хранит данные о постах, которые могут быть использованы для ведения записей или заметок.

Эта структура базы данных обеспечивает гибкость и масштабируемость системы управления складом, а также ведение журналов действий и управления пользователями.

**2.3 Разработка требований к безопасности**

В рамках разработки веб-приложения для управления складом и ведения заметок безопасность данных и пользователей является критически важным аспектом. Для этого необходимо детально разработать и внедрить механизмы, обеспечивающие надежную защиту приложения и его данных. В этом разделе будут рассмотрены основные требования к безопасности.

**Аутентификация:**

* **Использование Spring Security:** В качестве основного механизма аутентификации будет использован фреймворк Spring Security, который предоставляет мощные и гибкие инструменты для управления безопасностью.
* **Шифрование паролей:** Пароли пользователей будут храниться в зашифрованном виде с использованием алгоритмов хеширования, таких как BCrypt.

**Авторизация:**

* **Ролевое управление доступом:** Будет реализована система ролей и прав доступа, которая позволит управлять доступом пользователей к различным частям приложения на основе их роли. Например, администраторы будут иметь более широкие права по сравнению с обычными пользователями.
* **Тонкая настройка прав доступа:** В рамках каждой роли будет возможность детально настраивать права доступа к определенным действиям и данным, что обеспечит гибкость и безопасность системы.

**2.4 Разработка стратегии тестирования**

Тестирование является неотъемлемой частью процесса разработки программного обеспечения, направленной на обеспечение качества и надежности продукта. В проекте будут использоваться два основных метода тестирования: интеграционное и модульное тестирование.

**Модульное тестирование** (Unit Testing) — это метод тестирования, при котором отдельные компоненты (модули) программы проверяются на корректность работы путем изоляции каждого модуля и проверки его функциональности в изолированной среде.

**Интеграционное тестирование** (Integration Testing) — это метод тестирования, при котором проверяется взаимодействие между различными модулями или компонентами системы**.**

В рамках текущего проекта планируется внедрение методов тестирования, нацеленных на проверку функциональности сохранения объекта Item в базе данных, а также удаления этого объекта из базы данных.

**3. Практическая часть**

**3.1. Создание классов - сущностей**

Класс Item в данном случае представляет собой класс-сущность (entity class), используемый для маппинга данных из таблицы базы данных item на объекты Java в рамках ORM (Object-Relational Mapping) с использованием JPA (Java Persistence API).



**Аннотации**:

* @Getter и @Setter: Автоматически создают методы доступа (геттеры) и изменения (сеттеры) для всех полей класса.
* @Entity: Указывает, что данный класс является сущностью JPA и будет маппироваться на таблицу в базе данных.
* @Table(name = "item"): Указывает на имя таблицы в базе данных, к которой будет привязана эта сущность.
* @Id: Указывает, что это поле является первичным ключом.
* @GeneratedValue(strategy = GenerationType.IDENTITY): Автоматическое генерирование значений идентификатора с использованием стратегии идентичности.
* @Column(columnDefinition = "TEXT"): Указывает, что это поле будет хранить текстовые данные большой длины.
* @OneToMany(mappedBy = "item", cascade = CascadeType.REMOVE): Указывает на связь один-ко-многим с сущностью MyLog, при этом каскадное удаление логов при удалении товара.

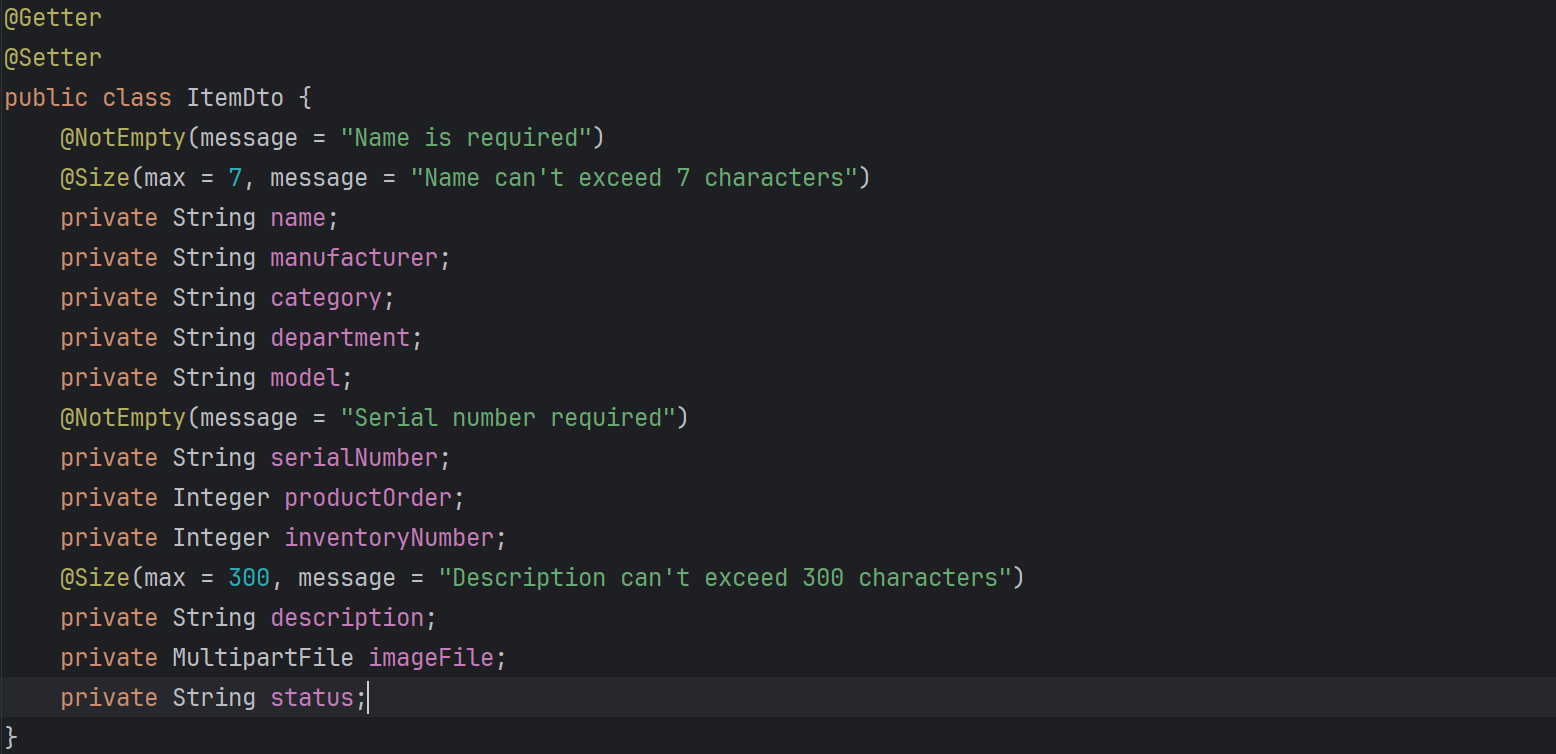
**Аналогично организованные классы**

Классы MyLog, Post, Role и User также структурированы для обеспечения простого и интуитивного управления данными в базе.

* **Основные сущности:**
  + **MyLog**: Поля для хранения информации о логах, связанных с действиями пользователей.
  + **Post**: Поля для хранения данных о заметках, таких как заголовок, содержание, даты создания и изменения.
  + **Role**: Поля для хранения информации о ролях пользователей, включая имя и id роли.
  + **User**: Поля для хранения информации о пользователях, такие как имя пользователя, пароль, полное имя и связанная информация.

Классы **DTO (Data Transfer Object)** являются важным элементом архитектуры приложений, позволяя эффективно передавать данные между различными слоями, такими как контроллеры, сервисы и представления. DTO-классы помогают отделить бизнес-логику от представления данных и упрощают валидацию и обработку входных данных. Кроме того, использование DTO-классов повышает безопасность приложения, поскольку позволяет ограничивать объем передаваемых между слоями данных, тем самым защищая внутренние структуры данных от несанкционированного доступа. В проекте используются несколько таких классов: ItemDto, PostDto, и UserDto. При работе с этими классам, будь то создание или редактирование, аннотации валидации выполняют контроль входных данных, гарантируя их соответствие заданным требованиям. Если данные не проходят валидацию, ошибки добавляются в объект BindingResult, который передается в контроллер. Контроллер проверяет наличие ошибок и, если они есть, возвращает пользователю форму с сообщениями об ошибках.

Класс **ItemDto:**



Валидация полей **ItemDto:**

**1.** Полеname*:*

* **@NotEmpty:** Не должно быть пустым. Сообщение об ошибке: "Name is required".
* **@Size(max = 7):** Длина значенияне должно превышать 7 символов. Сообщение об ошибке: "Name can't exceed 7 characters".

1. ПолеserialNumber*:*

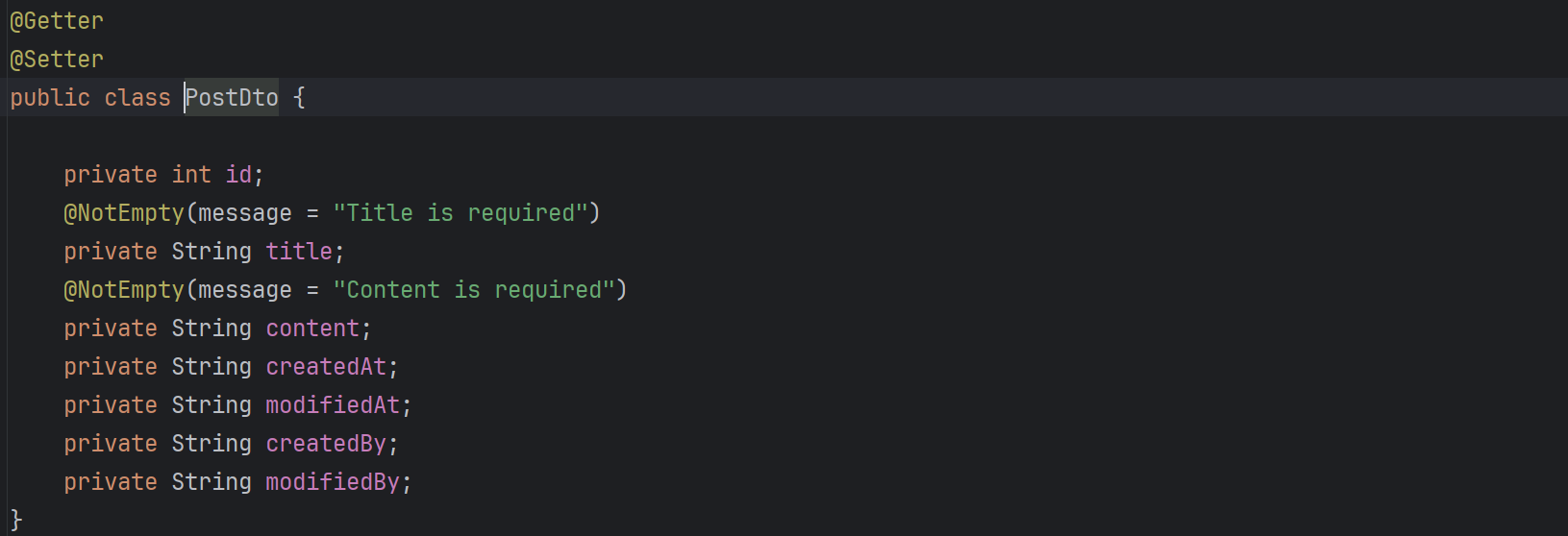
* **@NotEmpty:** Не должно быть пустым. Сообщение об ошибке: "Serial number required".

1. Полеdescription*:*

* **@Size(max = 300):** Не должно превышать 300 символов. Сообщение об ошибке: "Description can't exceed 300 characters".

Эти правила помогают гарантировать, что все поля заполнены корректно, а длина текста соответствует установленным ограничениям.

Класс **PostDto:**

****

Валидация полей **PostDto:**

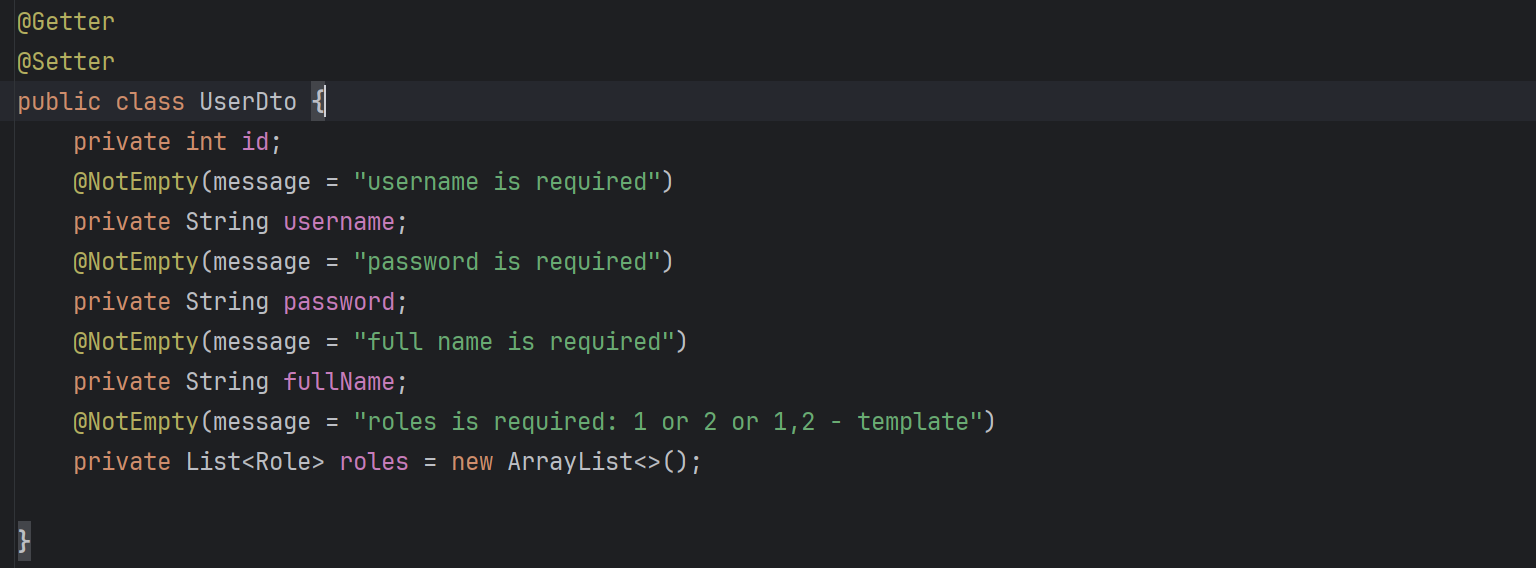
**1.** Полеtitle*:*

* **@NotEmpty:** Не должно быть пустым. Сообщение об ошибке: "Title is required".

**2.** Полеcontent*:*

* **@NotEmpty:** Не должно быть пустым. Сообщение об ошибке: "Content is required".

Класс **UserDto:**

****

Валидация полей **UserDto:**

**1.** Полеusername*:*

* **@NotEmpty:** Не должно быть пустым. Сообщение об ошибке: "username is required".

**2.** Полеpassword*:*

* **@NotEmpty:** Не должно быть пустым. Сообщение об ошибке: " password is required".

1. ПолеfullName*:*

* **@NotEmpty:** Не должно быть пустым. Сообщение об ошибке: "full name is required".

1. Полеroles*:*

* **@NotEmpty:** Не должно быть пустым. Сообщение об ошибке: "roles is required: 1 or 2 or 1,2 - template".

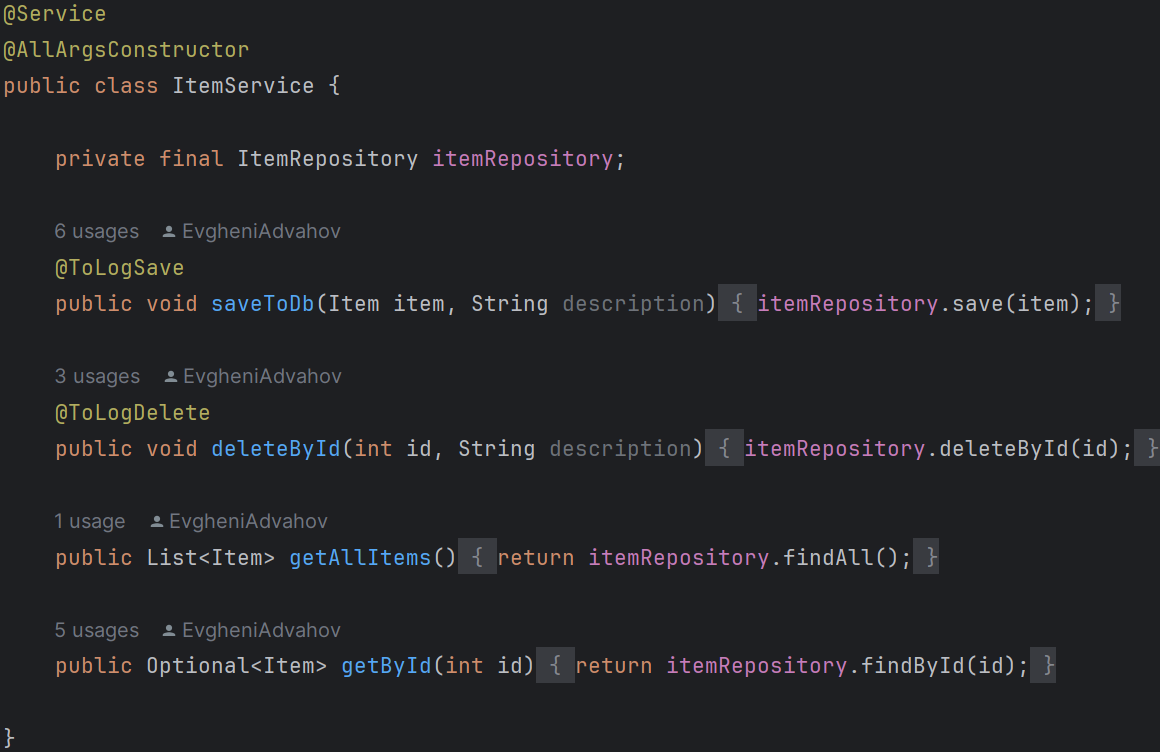
**3.2 Создание классов - сервисов**

Класс ItemService является сервисным слоем приложения, который предоставляет набор методов для работы с репозиторием ItemRepository. Он отвечает за бизнес-логику, связанную с управлением товарами, и обеспечивает взаимодействие между контроллерами и репозиторием. В этом классе инкапсулируются основные операции по управлению данными о товарах, такие как сохранение, обновление, удаление и получение информации о товарах.

**Основные преимущества ItemService:**

* Инкапсуляция бизнес-логики: Все операции с товарами сосредоточены в одном классе, что упрощает управление и модификацию бизнес-логики.
* Упрощение взаимодействия с репозиторием: Контроллеры обращаются к сервисному слою для выполнения операций, что делает код более чистым и понятным.
* Повышенная гибкость: Добавление новых операций или изменение существующих становится проще благодаря централизованному управлению бизнес-логикой.
* Логирование и обработка исключений: Сервисный слой позволяет легко добавлять логирование и обрабатывать исключения, что улучшает мониторинг и отладку приложения.

Класс **ItemService**:



**Аннотации:**

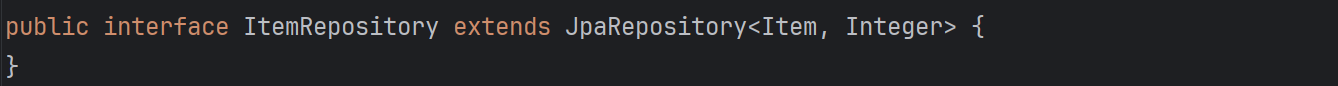
* **@Service -** указывает, что класс ItemService является сервисом Spring, что позволяет Spring автоматически обнаруживать его и управлять его жизненным циклом.
* **@AllArgsConstructor -** автоматически генерирует конструктор с параметрами для всех полей класса, облегчая создание экземпляров с заданными значениями.
* **@ToLogSave** и **@ToLogDelete -** кастомные аннотации, используются для

логирования операций.

Классы LogService, PostService и UserService организованы аналогичным образом, как и класс ItemService. Все эти классы являются сервисными компонентами в архитектуре приложения, предоставляющими методы для работы с соответствующими репозиториями и инкапсулирующими бизнес-логику, связанную с управлением данными.

**3.3 Создание интерфейсов - репозиториев**

Интерфейс **ItemRepository:**

****

ItemRepository представляет собой интерфейс, который предоставляет методы для выполнения операций CRUD. Этот интерфейс наследуется от JpaRepository, что позволяет автоматически получить набор стандартных методов для работы с базой данных без необходимости их ручного написания.

JpaRepository - это интерфейс из Spring Data JPA, который расширяет интерфейсы CrudRepository и PagingAndSortingRepository. Он предоставляет расширенные возможности для работы с базой данных, включая:

* **Стандартные методы CRUD**: Методы для сохранения, обновления, удаления и поиска сущностей.
* **Пагинация и сортировка**: Методы для работы с пагинацией и сортировкой результатов запросов.
* **Кастомные запросы**: Возможность определения своих методов с использованием JPQL или методов на основе именования.
* **Легкость в использовании**: Уменьшает количество шаблонного кода, необходимого для работы с базой данных, делая код чище и более понятным.

Репозитории LogRepository, PostRepository, RoleRepository и UserRepository организованы аналогичным образом, как и ItemRepository.

Таким образом, репозитории на основе JpaRepository обеспечивают удобный и эффективный способ работы с данными, минимизируя усилия разработчиков и повышая продуктивность.

**3.4 Создание классов - котроллеров**

В Java Spring контроллеры являются центральным элементом модели MVC (Model-View-Controller). Они управляют взаимодействием между пользователем и приложением, обрабатывая HTTP-запросы, вызывая соответствующие методы сервисов, и определяя, какое представление (View) должно быть возвращено пользователю.

**Основные функции контроллёров:**

* 1. **Обработка HTTP-запросов**: Контроллеры принимают и обрабатывают HTTP-запросы (GET, POST, PUT, DELETE и т.д.). Они определяют, какой метод должен быть вызван для обработки каждого запроса.
  2. **Выбор представления (View)**: После обработки запроса контроллер выбирает соответствующее представление, которое должно быть возвращено пользователю. Это может быть HTML-страница, JSON-ответ или другой тип данных, адаптированный под запрос.
  3. **Обмен данными с моделью (Model)**: Контроллеры взаимодействуют с моделью, чтобы получить или обновить данные. Они вызывают методы сервисов и репозиториев для выполнения операций с данными, обеспечивая корректную работу бизнес логики.

Ниже представлен класс **ItemController***.*

****

**Аннотации:**

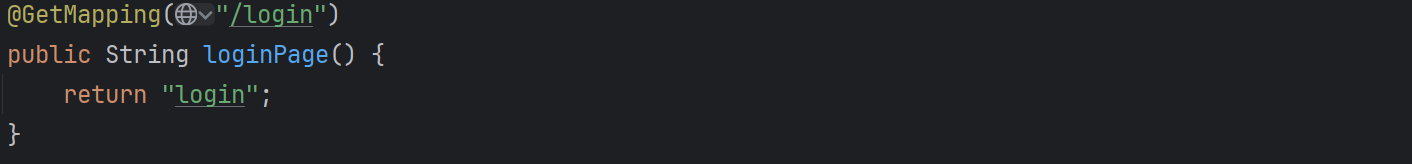
* **@Controller -** Указывает, что этот класс является контроллером Spring MVC.
* **@RequestMapping("/") -** Определяет URL-адрес, по которому будут обрабатываться запросы этим контроллером. В данном случае указан корневой URL ("/"), что означает, что все запросы к корню веб-приложения будут обрабатываться методами этого контроллера.
* **@AllArgsConstructor**: - Автоматически генерирует конструктор с одним параметром для каждого поля в классе. Упрощает создание экземпляров класса, автоматически создавая конструктор, который принимает все аргументы, необходимые для инициализации полей класса.

**Поля и зависимости**

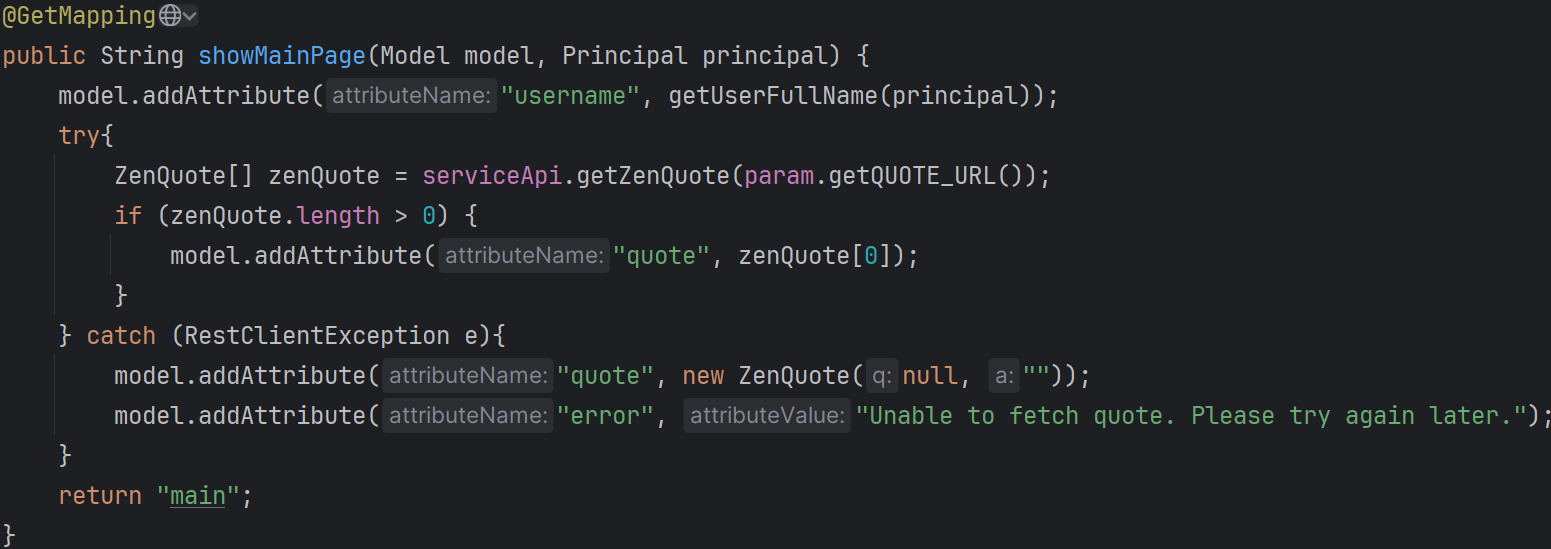
* **itemService:** Сервис для работы с товарами.
* **userService:** Сервис для работы с пользователями.
* **logService:** Сервис для работы с логами.
* **serviceApi:** Сервис, который использует RestTemplate для выполнения HTTP-запросов и получения данных из внешнего источника.
* **param:** Параметры конфигурации(configuration-processor) приложения.
* **Счетчики Prometheus:** *itemsCounterAdded* и *itemsCounterRemoved* используются для мониторинга количества добавленных и удаленных товаров.

**Методы контроллера:**

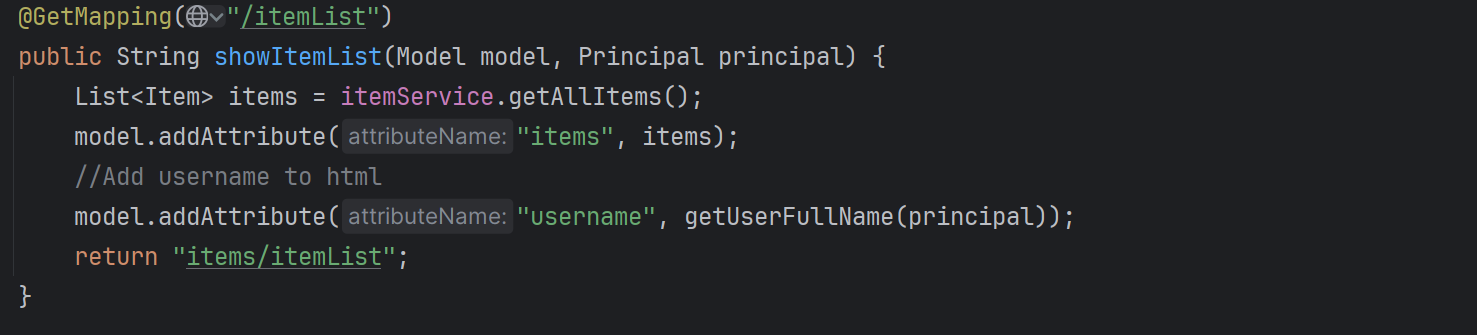
* **loginPage:** Отображение страницы входа**.**

****

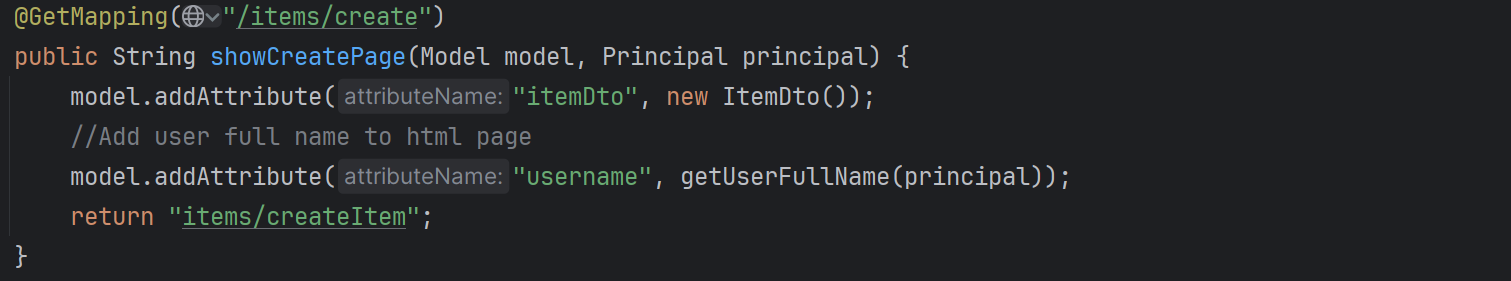
* **showMainPage:** Этот метод извлекает данные о пользователе и цитату для отображения на главной странице.



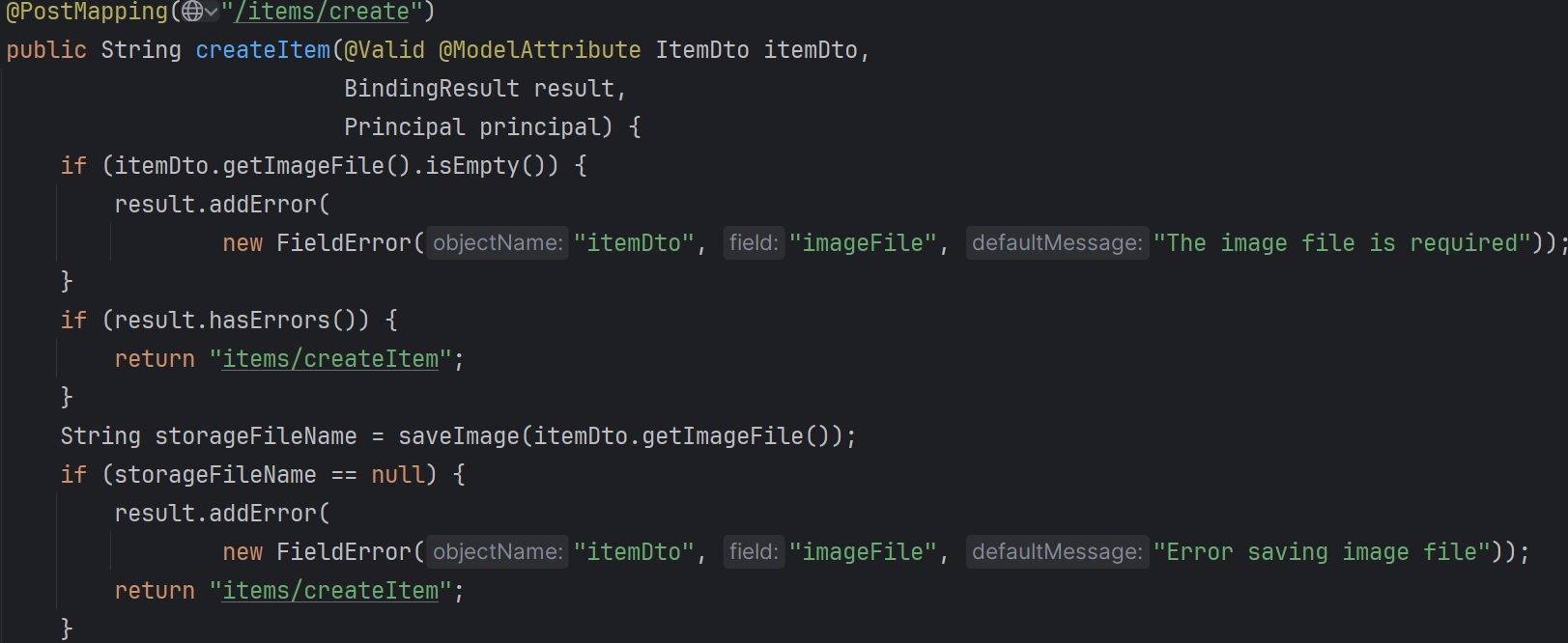
* **showItemList:** Метод showItemList отображает страницу со списком всех товаров. Этот метод извлекает все товары из базы данных, добавляет их в модель, а также добавляет имя текущего аутентифицированного пользователя для отображения в представлении.

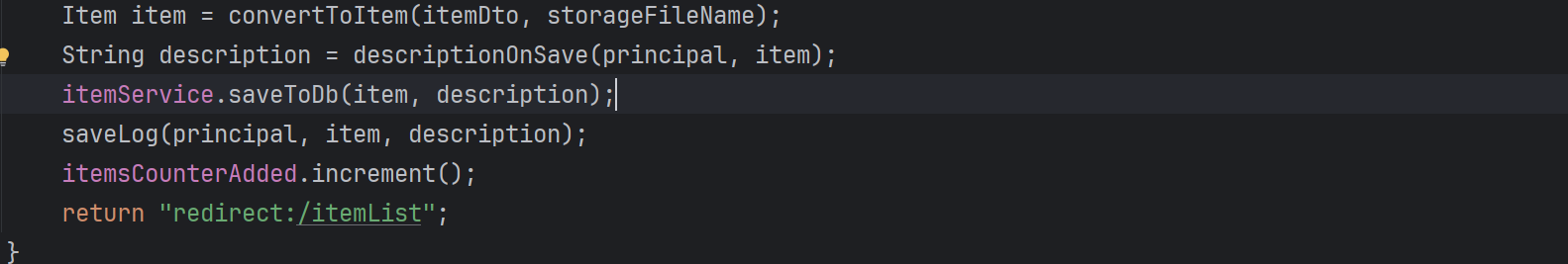
****

* **showCreatePage:** Отображение страницы создания товара. Объект ItemDto передаётся в метод для инициализации формы на странице, чтобы пользователь мог ввести данные для нового элемента.

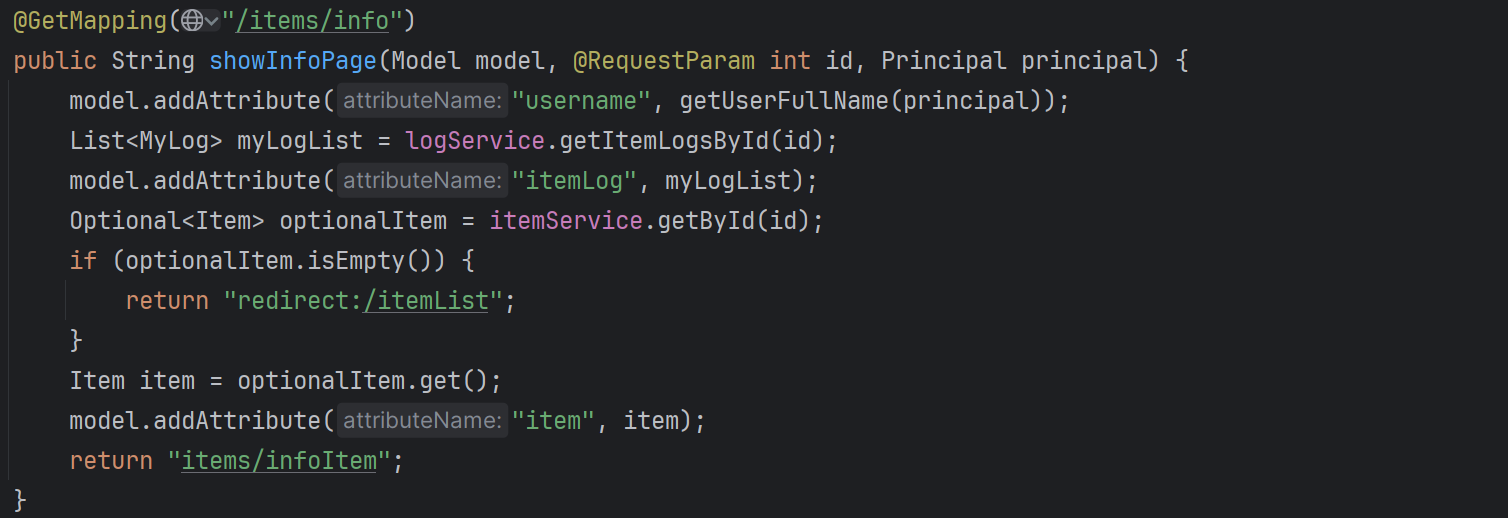
****

* **createItem:** Этот метод обрабатывает POST-запросы по адресу */items/create* для создания и сохранения информации о товаре. В этом методе используется объект ItemDto для передачи данных о товаре из представления в контроллер. Аннотация @ModelAttribute указывает на то, что объект itemDto должен быть инициализирован значениями из параметров запроса. Аннотация @Valid указывает на валидацию объекта согласно правилам в классе ItemDto. После проверки данных на ошибки, если они есть, возвращается страница создания товара с сообщениями об ошибках. В противном случае изображение товара сохраняется, объект Item создаётся на основе данных из ItemDto, и он сохраняется в базу данных. После сохранения товара выполняются другие действия, такие как сохранение логов и увеличение счетчика товаров. Метод возвращает страницу со списком товаров.

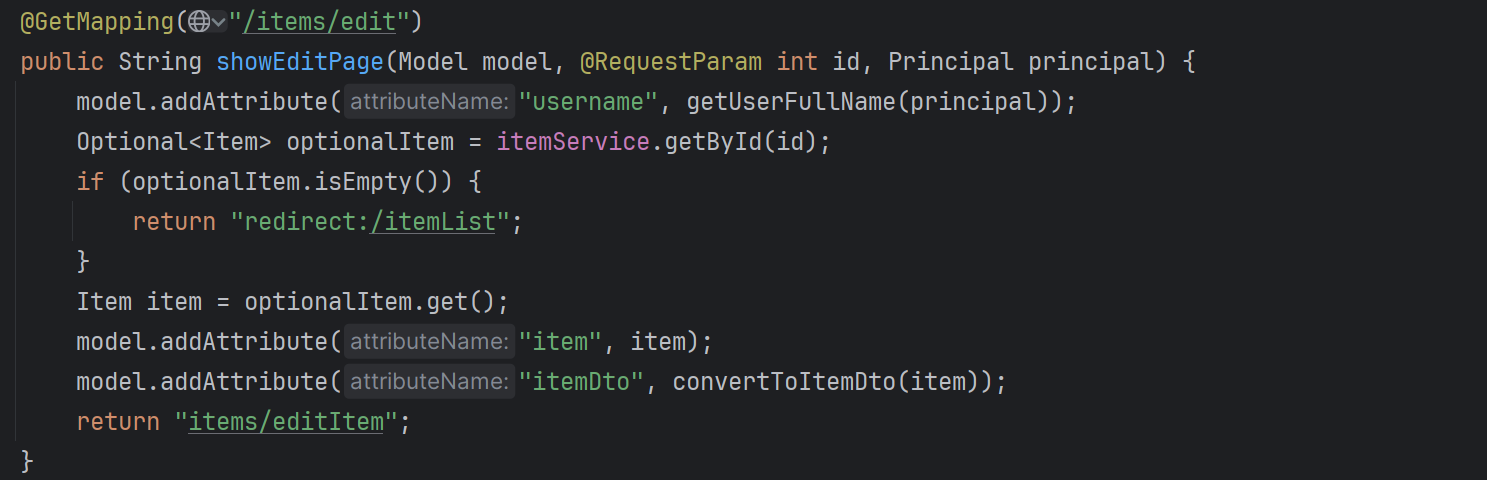
****

****

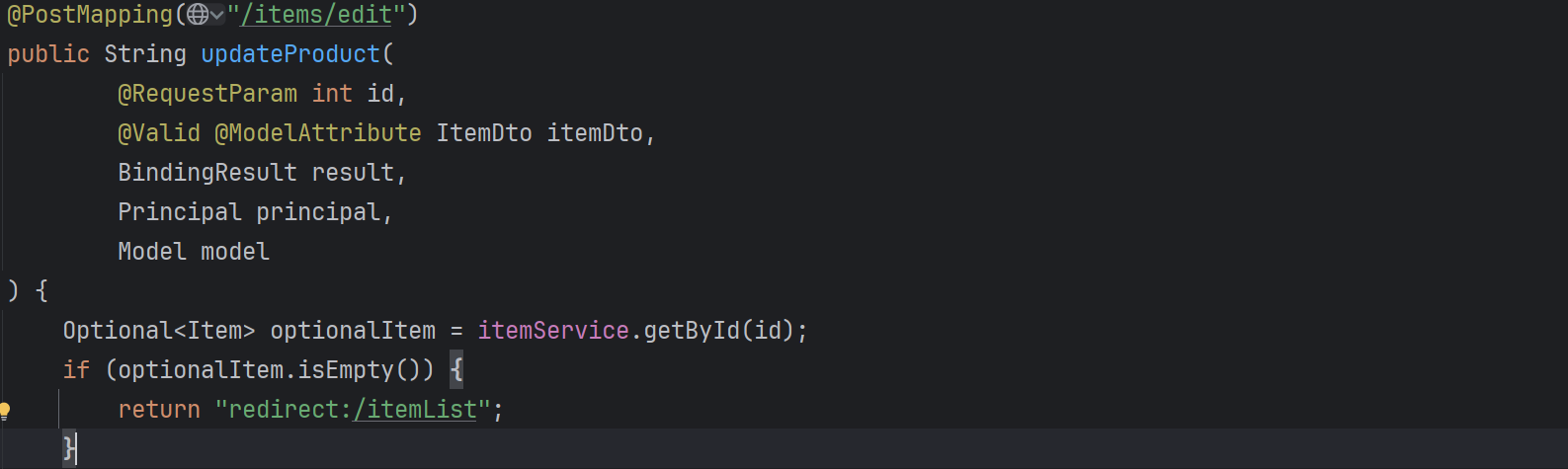
* **showInfoPage:** Отображение информации о товаре.

****

* **showEditPage:** Отображение страницы редактирования товара. Этот метод обрабатывает GET-запросы по адресу *"/items/edit"*. Он принимает три параметра: объект Model для передачи данных в представление, id элемента для редактирования и объект Principal для аутентификации пользователя.

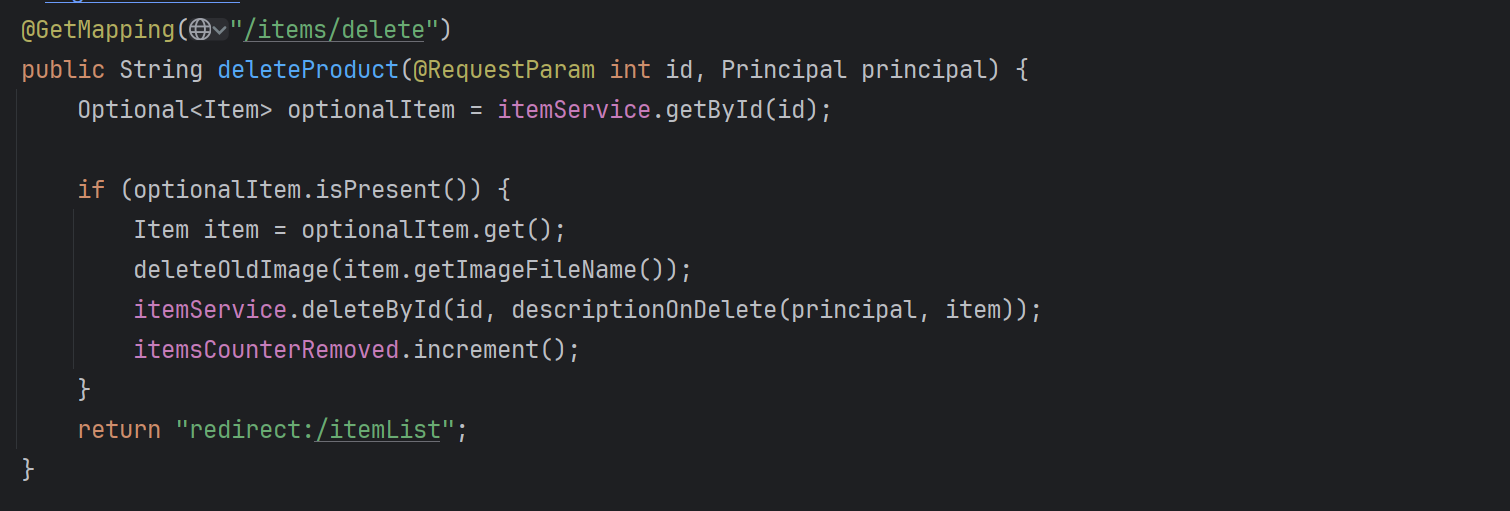
****

* **updateProduct:** Метод обрабатывает POST-запросы по адресу */items/edit*. Он получает параметр id для определения товара, который нужно отредактировать, а также данные о товаре из объекта ItemDto. Метод выполняет валидацию и, при наличии ошибок, возвращает страницу редактирования с сообщениями об ошибках. Затем метод получает объект товара по его id из базы данных. Если такого товара не существует, происходит перенаправление на страницу со списком товаров. Если загружено новое изображение, старое удаляется, а новое сохраняется. Далее обновляется информация о товаре и сохраняется в базе данных. Если произошли изменения в описании товара, они также сохраняются в логах. Метод возвращает страницу со списком товаров.



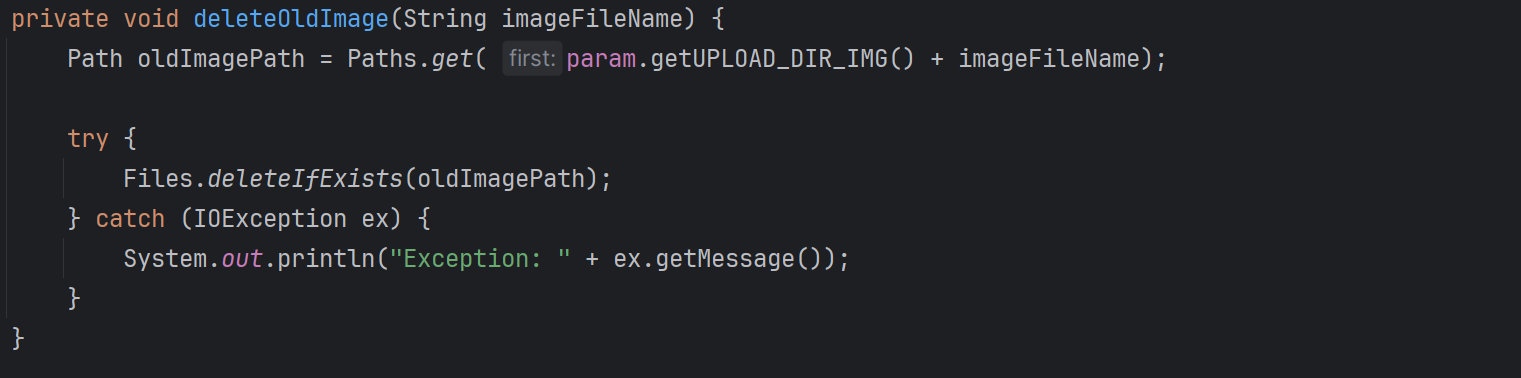


* **deleteProduct:** Удаление товара.

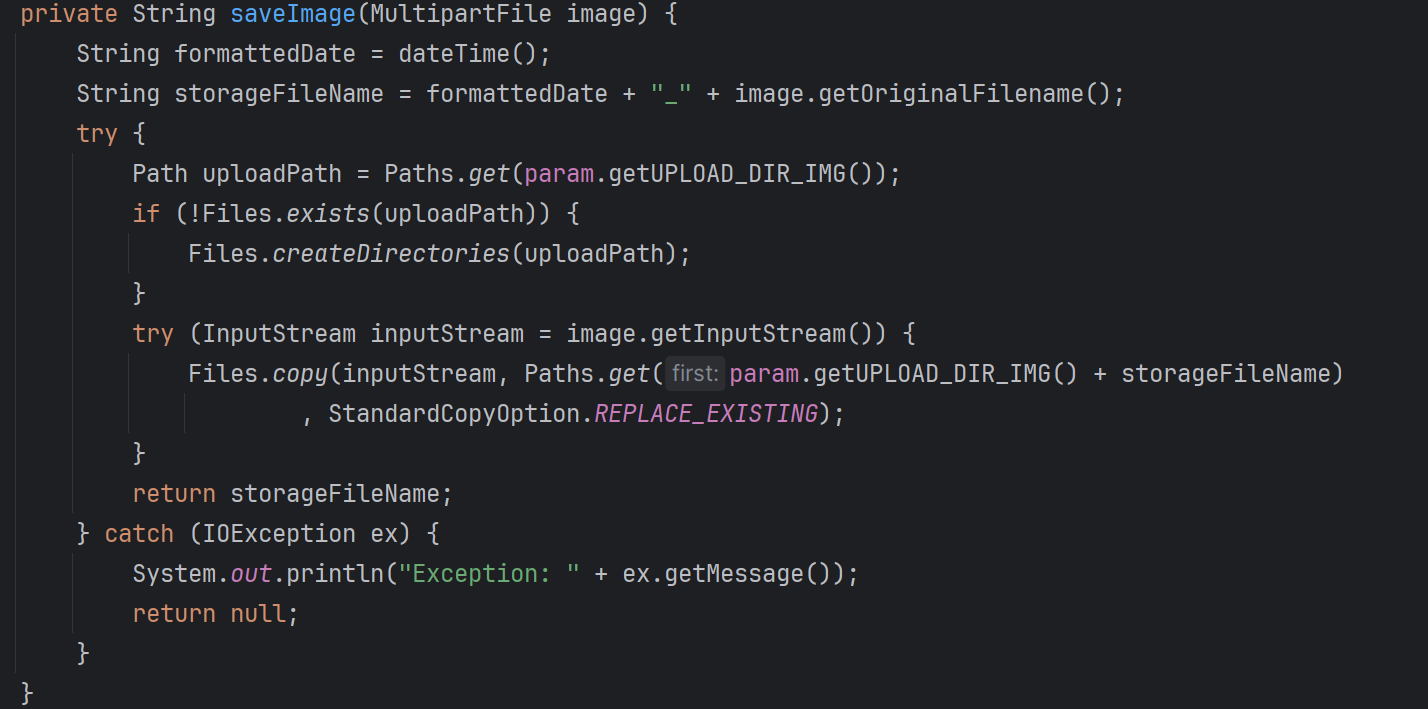
****

### Вспомогательные методы :

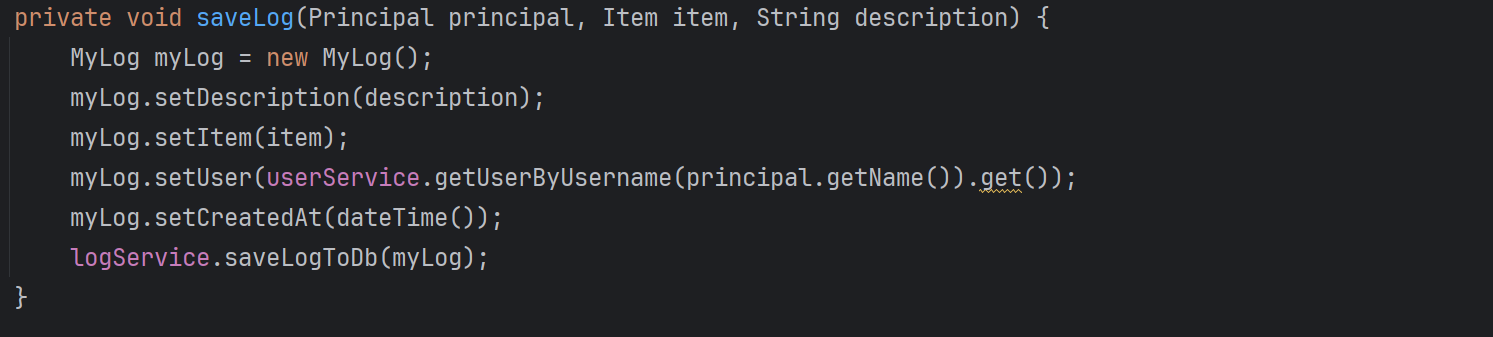
### deleteOldImage: Удаление старого изображения при редактировании и удалении объекта.



* **saveImage:** Сохранение изображения.



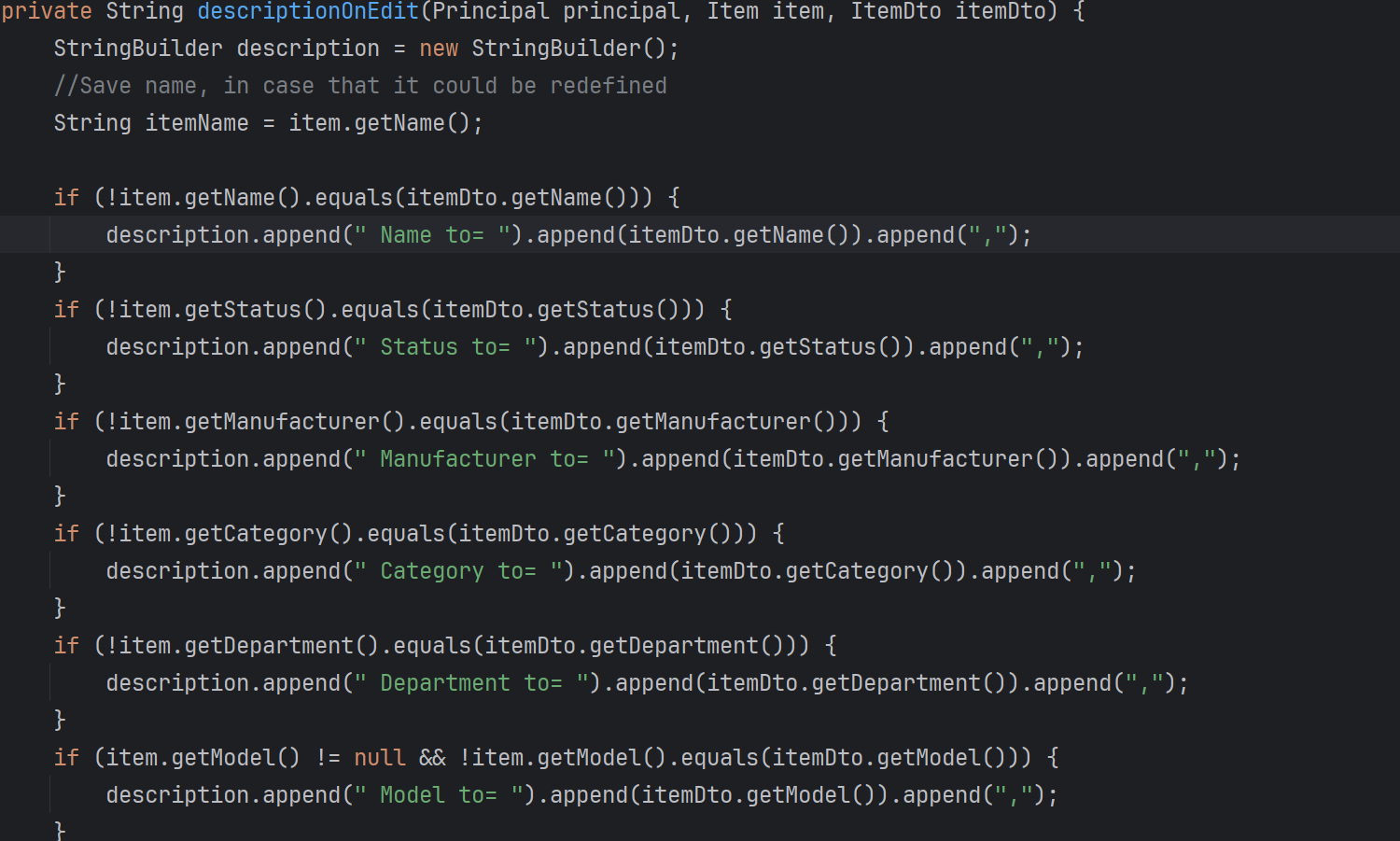
### saveLog: Сохранение лога.

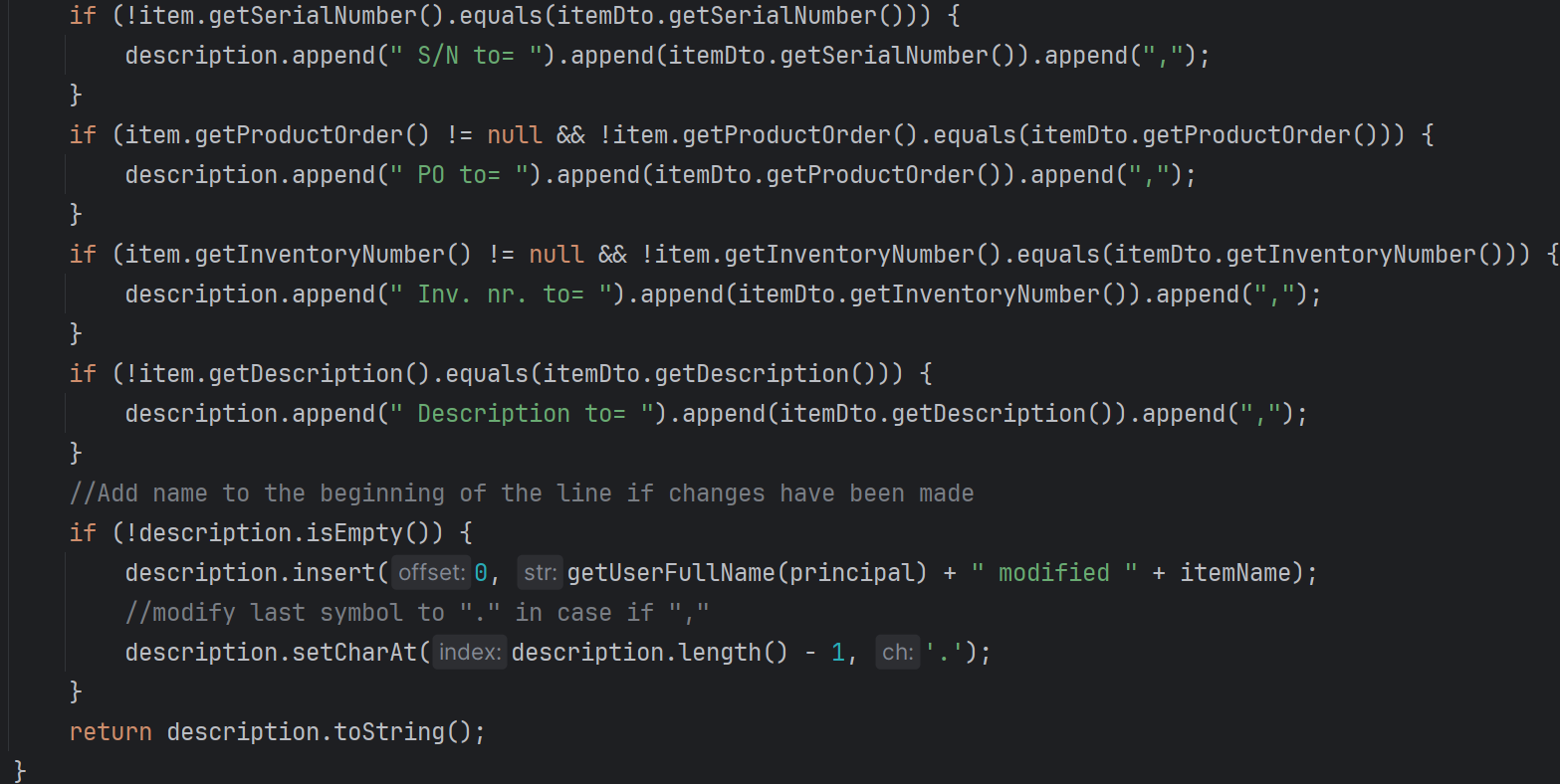


### descriptionOnSave: Создание описания при сохранении.

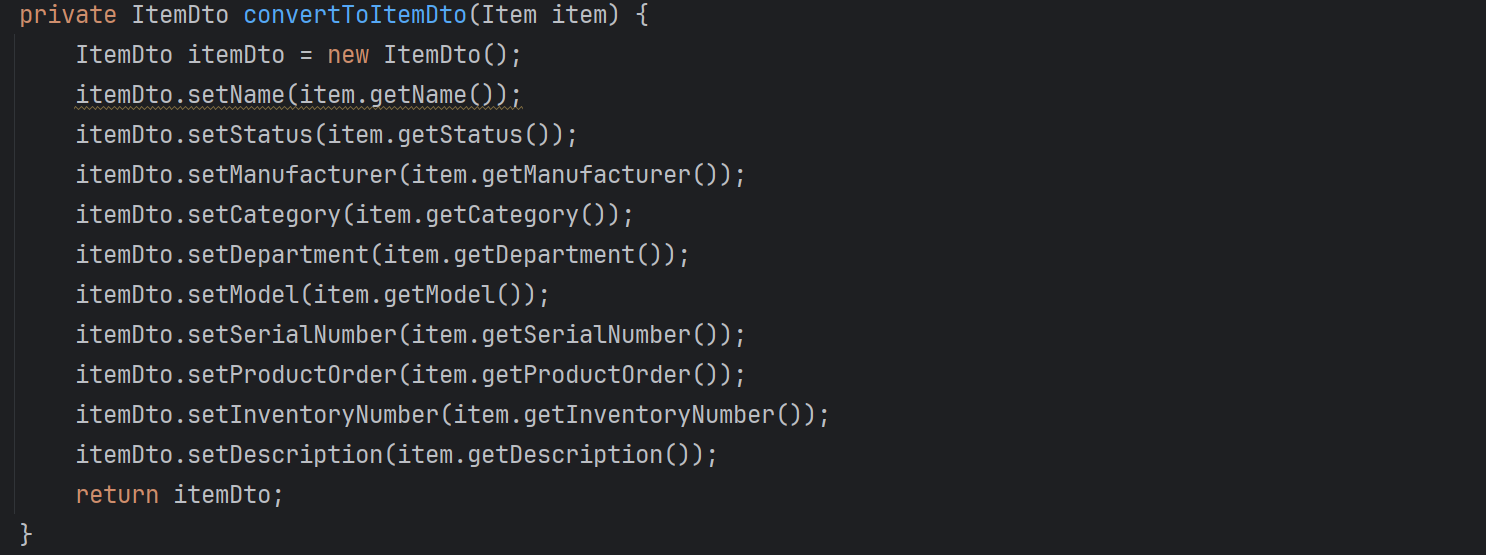


* **descriptionOnEdit:** Метод генерирует описание изменений, внесённых в объект item. Он сравнивает значения полей объекта item с соответствующими значениями в объекте itemDto и формирует строку, описывающую те поля, в которых были сделаны изменения.

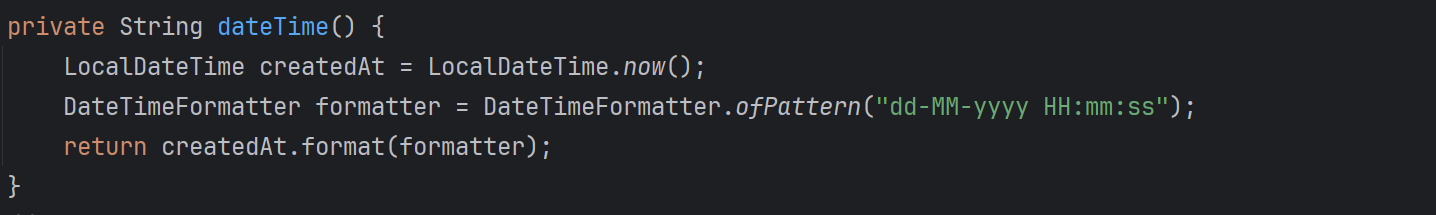




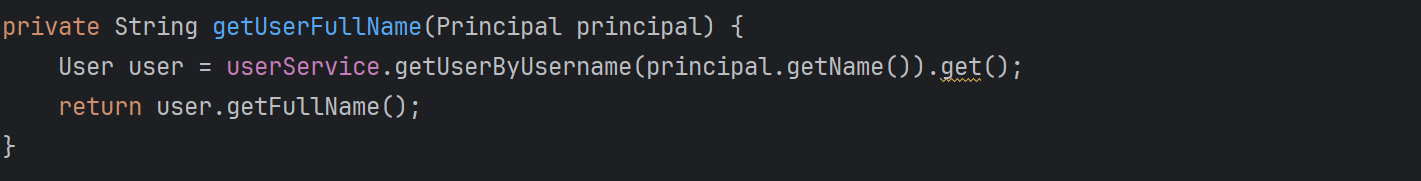
* **сonvertToItemDto:** Преобразование объекта **Item** в **DTO.**

****

* **dateTime:** Получение текущей даты и времени.



* **getUserFullName**: Получение полного имени пользователя.



Этот контроллер обеспечивает полный цикл операций CRUD (Create, Read, Update, Delete) для товаров, а также логирование действий пользователей и мониторинг через Prometheus.

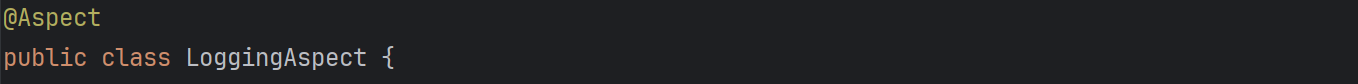
PostController и UserController устроены и работают аналогичным образом.

**3.5 Логирование**

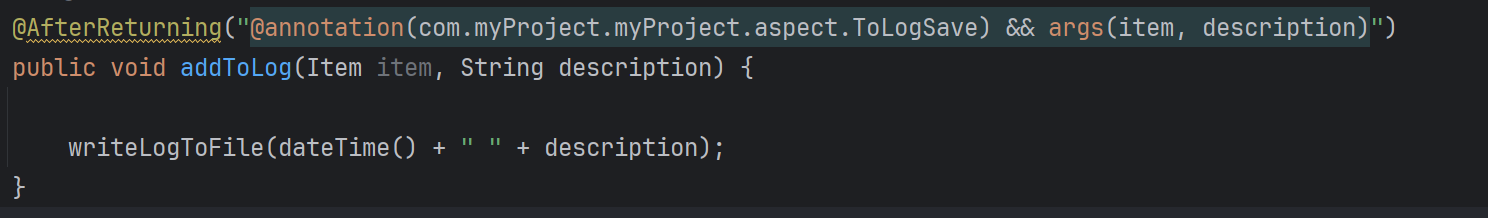
Логирование в данном проекте осуществляется двумя способами: с использованием аспектов и с помощью базы данных MySQL.

1. **C использованием Аспектов:**

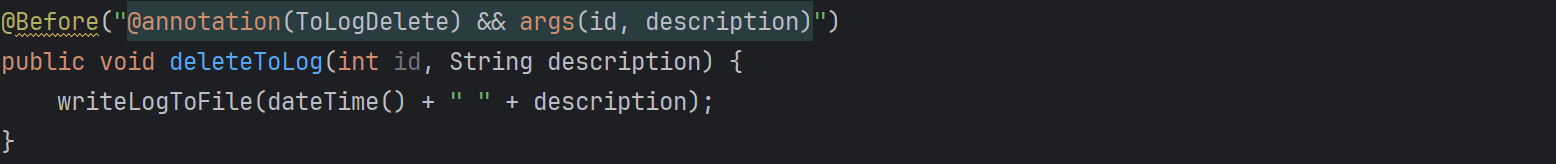
* Создан класс с аннотацией @Aspect:



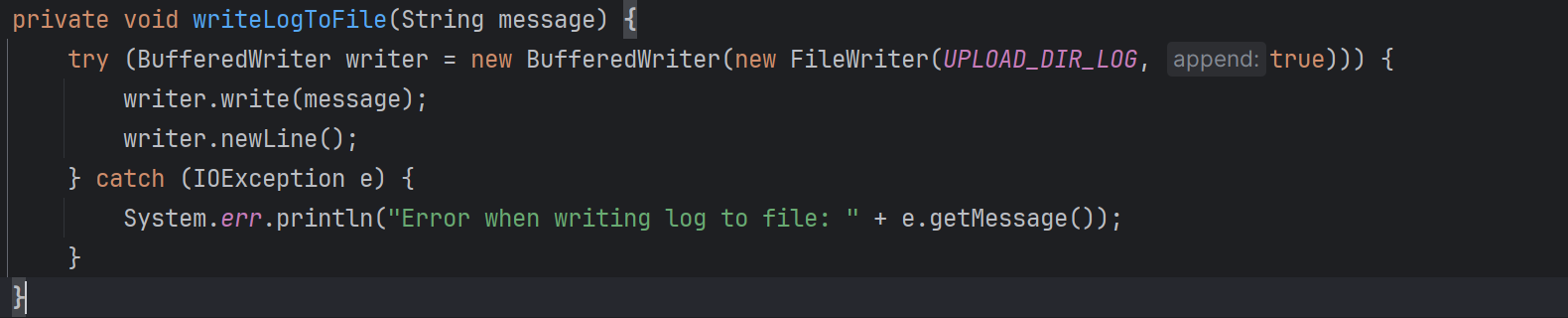
Аннотация @AfterReturning указывает что, метод addToLog должен выполняться после успешного завершения метода, аннотированного @ToLogSave. Метод addToLog принимает два параметра: item и description, а затем записывает description в лог-файл.

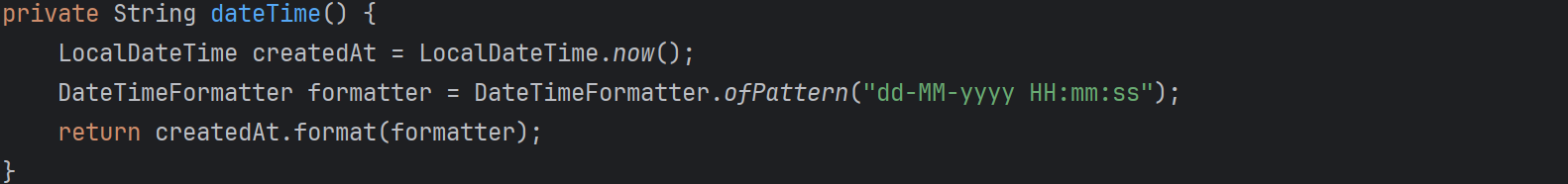


Аннотация @Before указывает, что метод deleteToLog должен выполняться перед методом, аннотированным @ToLogDelete. Метод deleteToLog принимает два параметра: id и description, а затем записывает description в лог-файл.



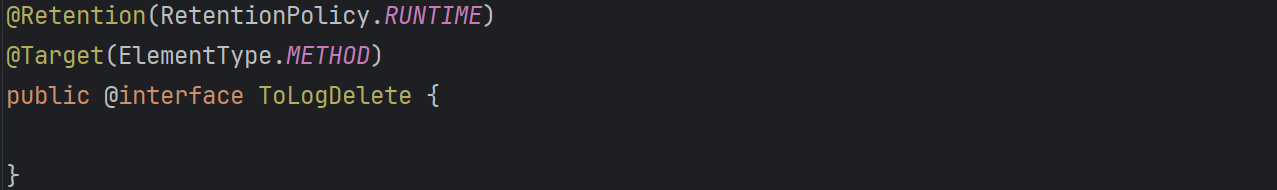
Метод writeLogToFile записывающий лог в файл:



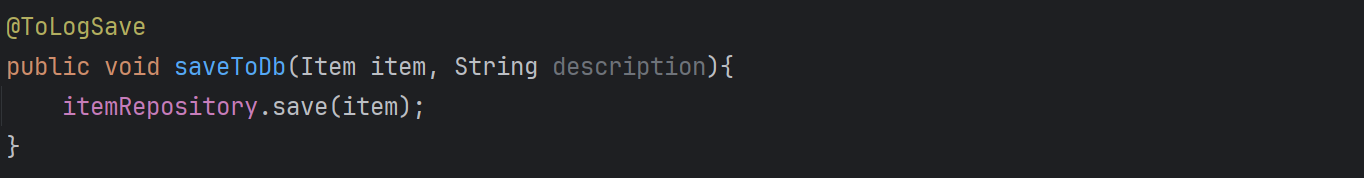
Метод dateTime() для получения текущей даты и времени:

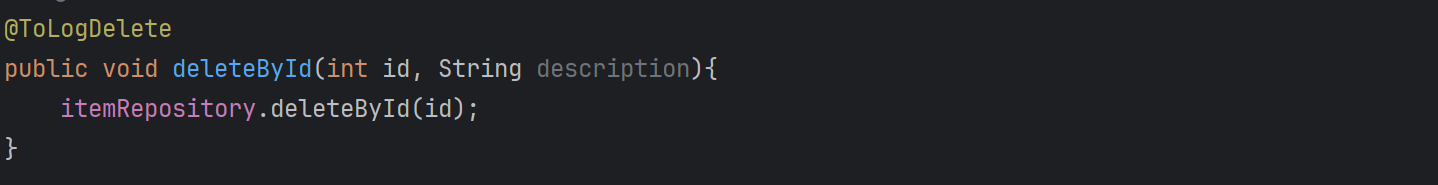
* Создание аннотаций @ToLogDelete и @ToLogSave



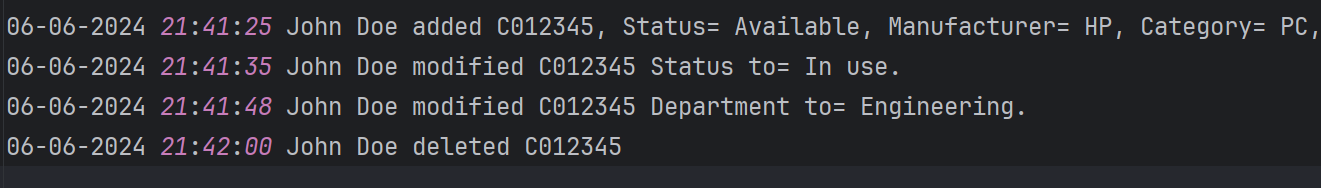


* Применение аннотаций @ToLogSave и @ToLogDelete



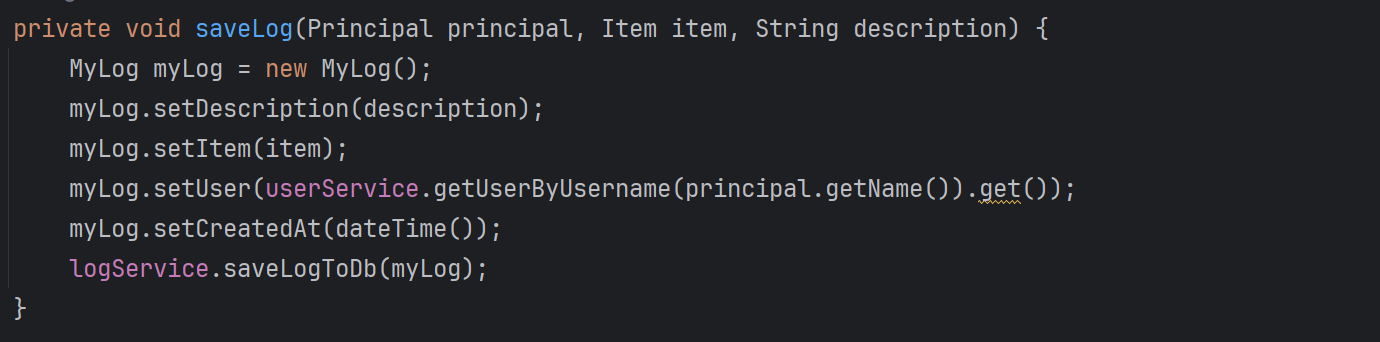


* Пример лога:

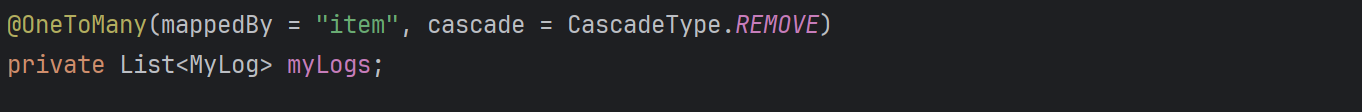


1. **С помощью базы данных MySQL:**

ItemController использует метод saveLog при создании или изменении объекта Item.



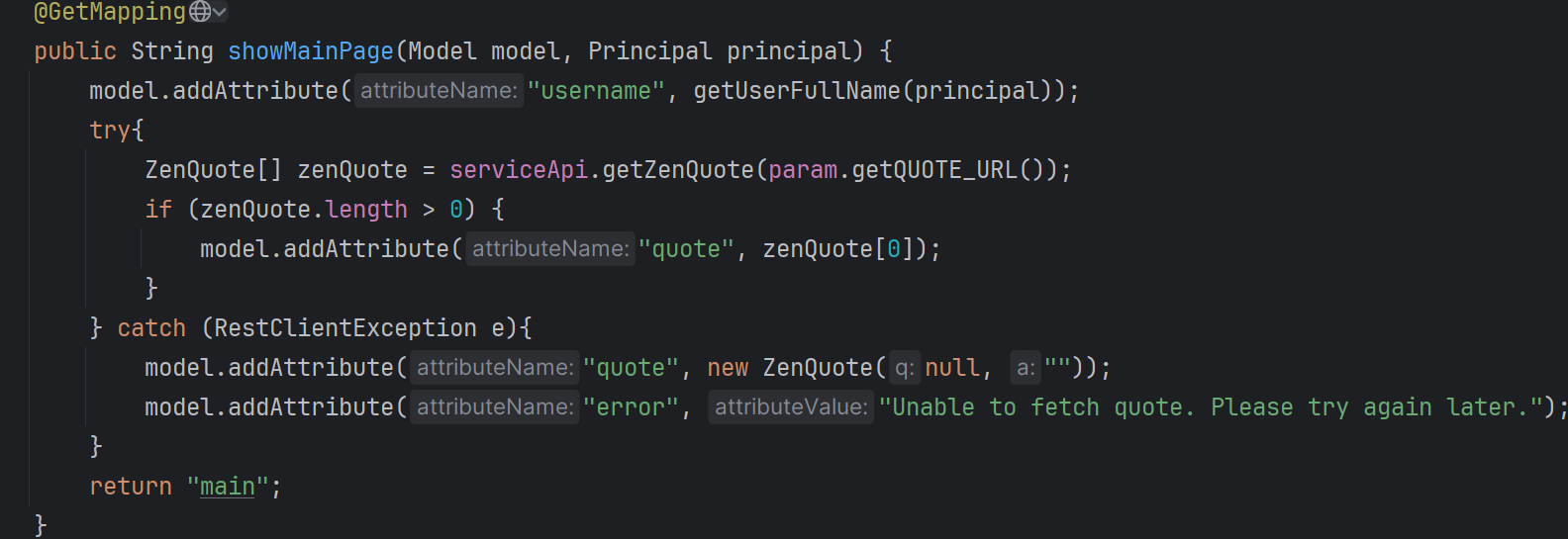
При удалении объекта из базы данных также происходит удаление всех связанных с ним логов. Это сделано для того, чтобы в базе данных оставались только логи тех объектов, которые существуют. Этот механизм реализован через каскадное удаление.



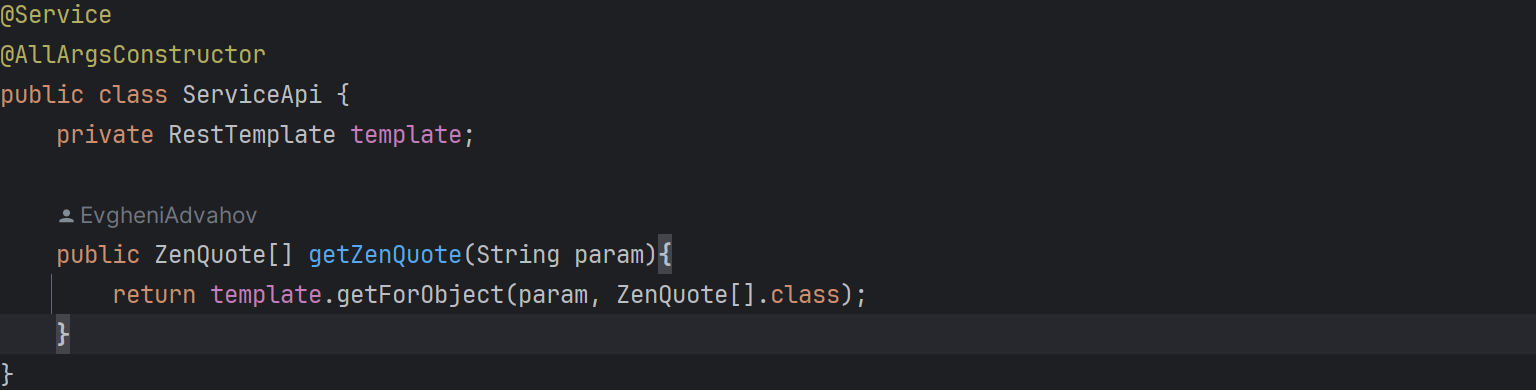
**3.6 Подключение внешнего API ZenQuotes.io**

Zenquotes.io представляет собой бесплатный API, который предоставляет цитаты для вдохновения.

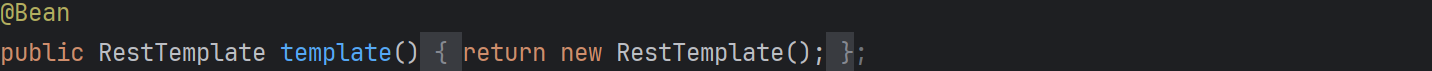
* Подключение этого Api в контроллере ItemController. При каждом обновлении главной страницы - новая цитата.



* Используем класс ServiceApi для изолирования внешних сервисов:



* Подключаем RestTemplate:



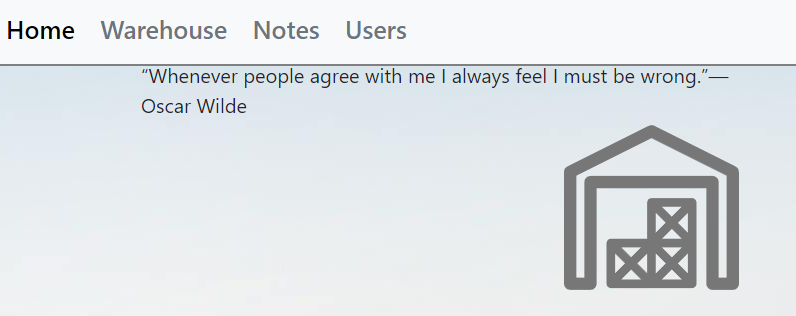
Основные функции Restemplate это:

1. отправка HTTP-запросов , таких как GET, POST, PUT, DELETE и т.д.
2. автоматически сериализует объекты Java в формат JSON или XML при отправке запросов и десериализует ответы обратно в объекты Java.
3. Встроенные механизмы для обработки HTTP-ошибок и позволяет настраивать обработчики ошибок.

* Код страницы main для отображения цитаты:



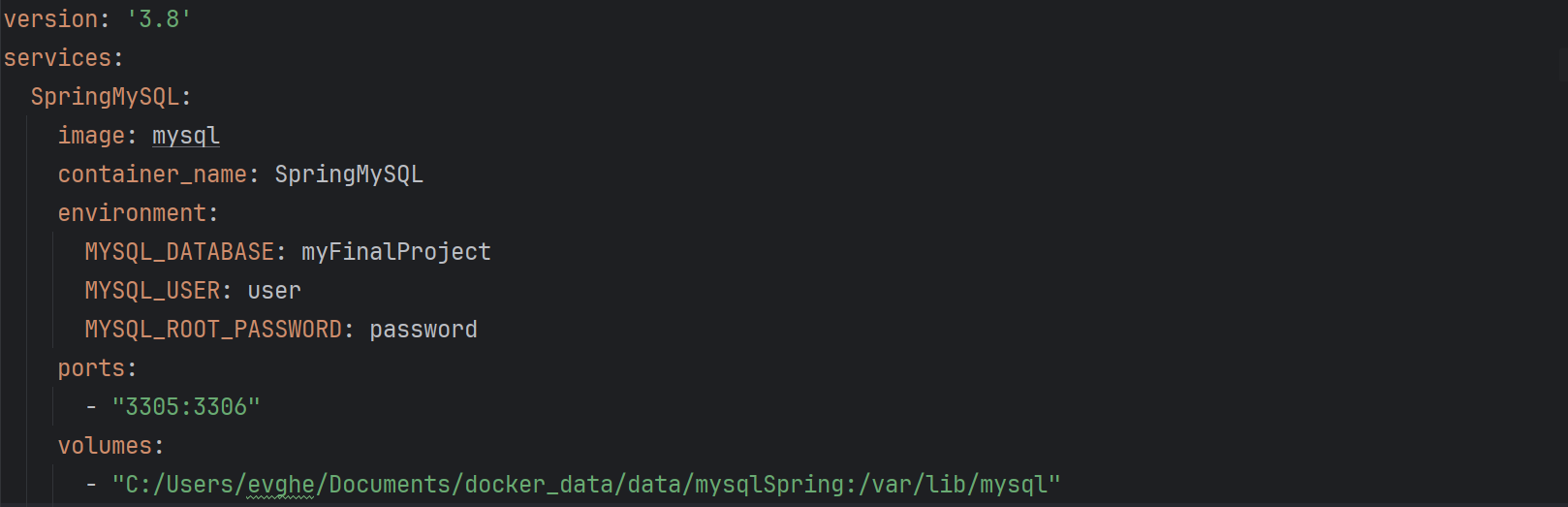
* Пример отображения:

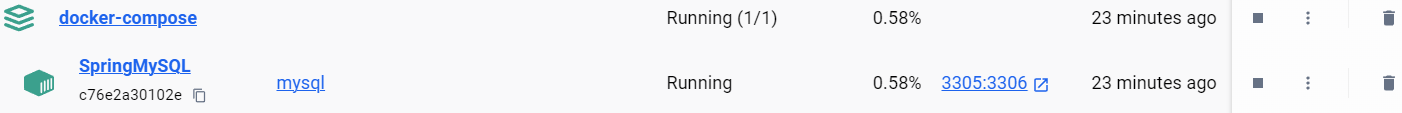


**3.7 Разворачивание Docker контейнера и подключение базы данных MySQL**

Запуск конфигурационного файла docker-compose.yml:

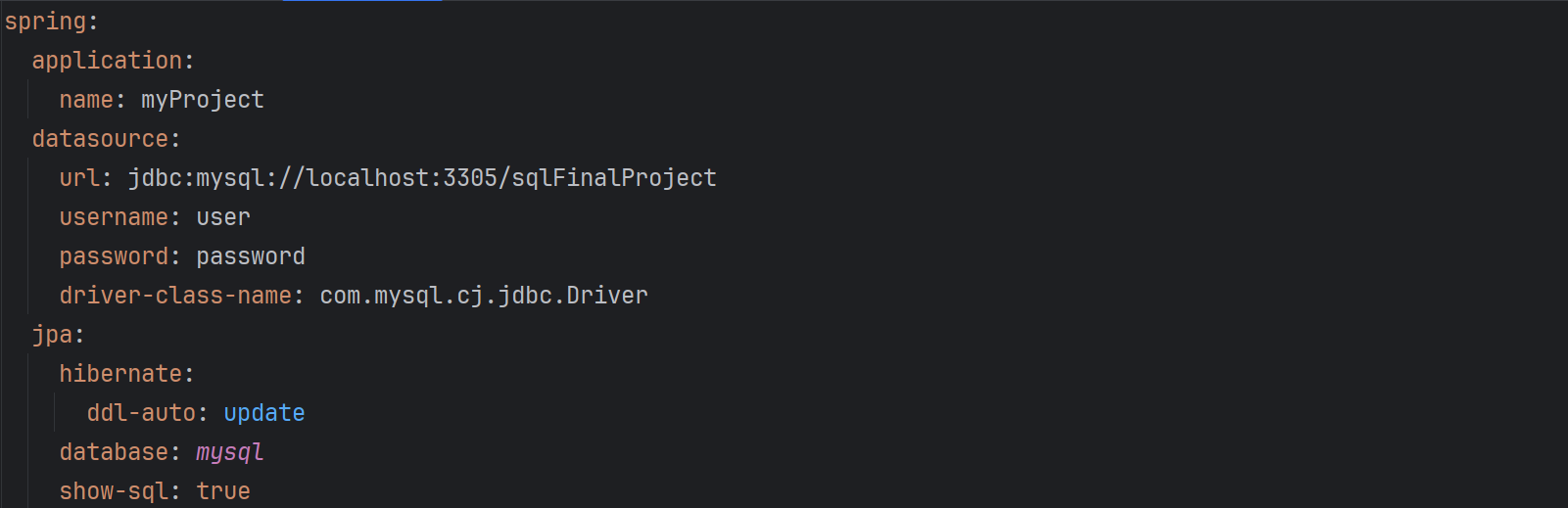
* + docker compose up



****

Подключение базы данных MySQL:

Настройки файла properties.yml:



**3.8 Подключение конфигурационного процессора**

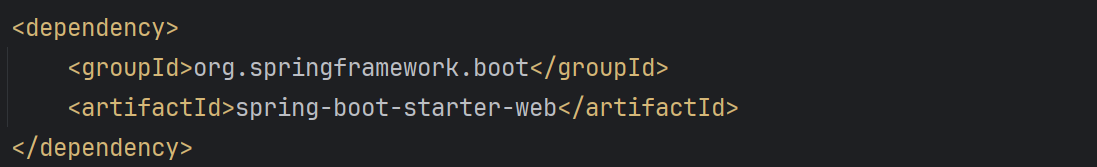
Конфигурационный процессор в Java, часто упоминаемый как configuration-processor, используется для обработки аннотаций конфигурации в проектах Spring Boot. Этот инструмент генерирует метаданные конфигурации, которые помогают в автозаполнении и валидации конфигурационных параметров в IDE, таких как IntelliJ IDEA или Eclipse.

**Основные функции конфигурационного процессора:**

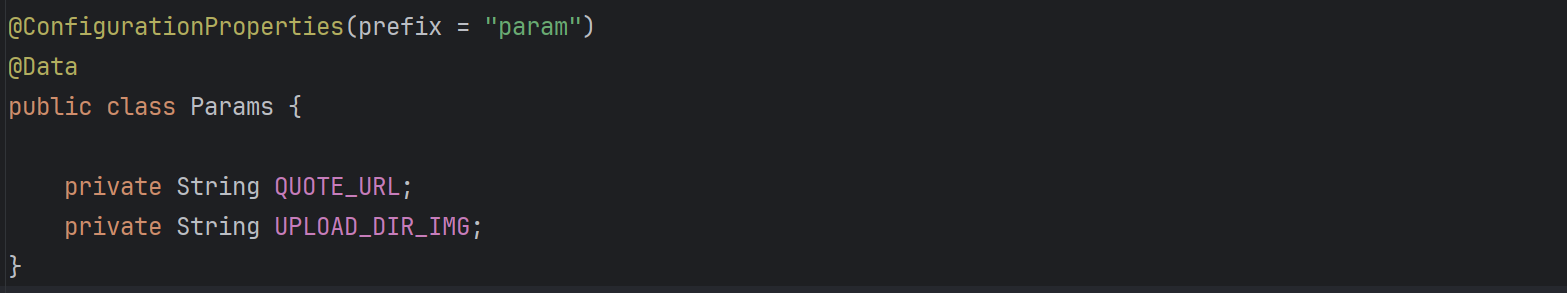
1. Генерация метаданных: Процессор генерирует файл META-INF/spring-configuration-metadata.json, содержащий информацию о всех параметрах конфигурации, определенных в вашем приложении. Это позволяет IDE предложить автозаполнение и валидацию при редактировании файлов конфигурации, таких как application.properties или application.yml.
2. Поддержка типизированных свойств: Когда вы используете аннотации типа @ConfigurationProperties, процессор помогает обеспечить типобезопасность и удобство работы с этими свойствами

Подключение:

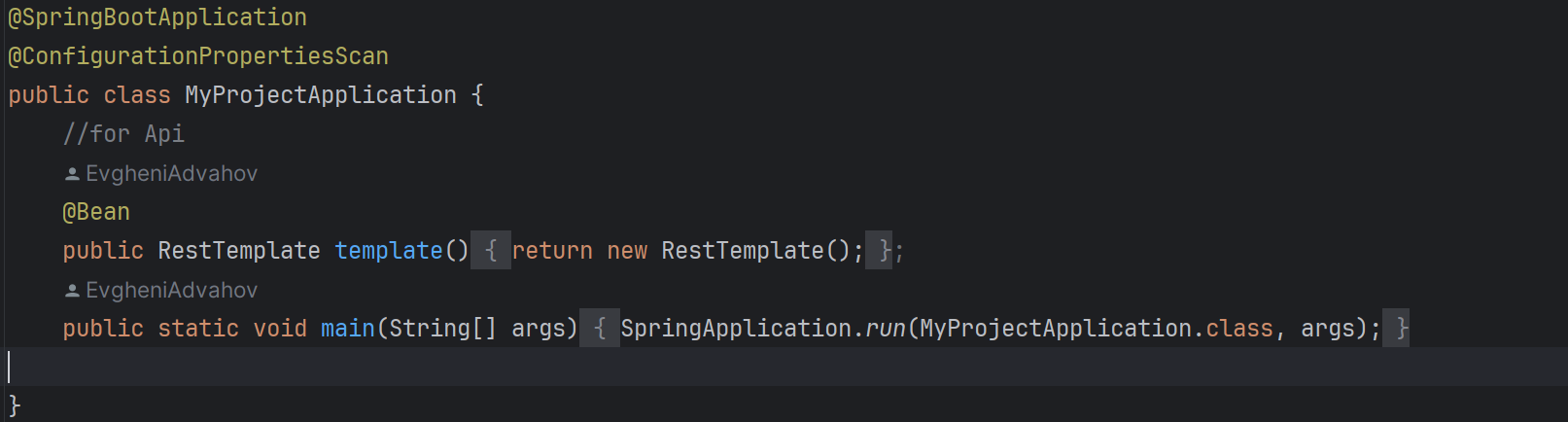
* 1. Для подключения нужна зависимость spring-boot-configuration-processor. В данном проекте spring-boot-starter-web включает в себя эту зависимость.



* 1. Создание класса с конфигурацией:



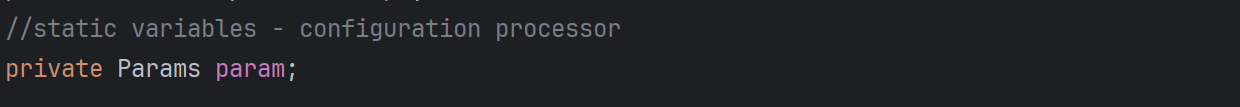
* 1. Регистрация класса конфигурации @ConfigurationPropertiesScan:



* 1. Конфигурация в configuration.yaml:



* 1. Использование переменных в контроллере ItemController:







Применение configuration-processor помогает избежать использования "магических" строковых значений прямо в коде и упрощает процесс изменения конфигурационных параметров. Это достигается за счет вынесения этих параметров в файлы конфигурации, такие как application.properties или application.yml.

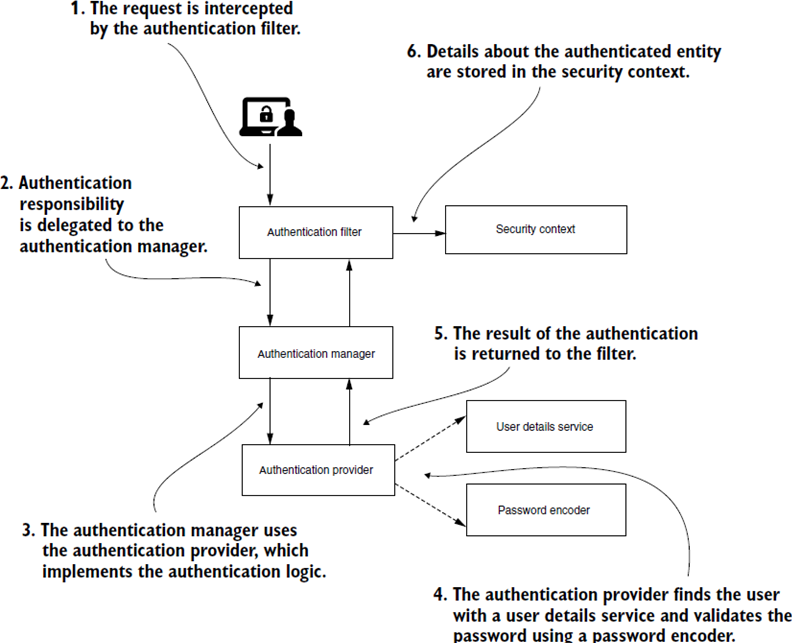
**3.9** **Безопасность**

Spring Security - это мощный и гибкий фреймворк для обеспечения аутентификации, авторизации и защиты приложений на основе Spring. Он предоставляет разнообразные функции и инструменты для обеспечения безопасности веб-приложений, REST API и других приложений, построенных на платформе Spring.

Основные возможности и концепции Spring Security:

1. Аутентификация: Spring Security предоставляет множество способов аутентификации, включая форму входа, базовую аутентификацию, аутентификацию по токену и другие. Он также поддерживает расширяемость через собственные механизмы аутентификации и интеграцию с внешними системами аутентификации, такими как OAuth2.
2. Авторизация: Фреймворк предоставляет механизмы авторизации для контроля доступа к различным частям приложения на основе ролей, разрешений и других атрибутов безопасности.
3. Фильтрация запросов: Spring Security интегрируется в цепочку фильтров Spring MVC для обработки запросов и применения правил безопасности к ним. Это позволяет легко контролировать доступ к ресурсам и обеспечивать безопасность на уровне HTTP-запросов.
4. Защита от атак: Фреймворк предоставляет защиту от таких атак, как CSRF (межсайтовое подделывание запроса), инъекции, сессионные атаки и другие. Встроенные механизмы защиты помогают минимизировать уязвимости и улучшить общую безопасность приложения.
5. Интеграция с другими модулями: Spring Security легко интегрируется с другими модулями Spring, такими как Spring MVC, Spring Boot, Spring Data и другими. Это позволяет разработчикам быстро и легко добавлять функциональность безопасности в свои приложения.
6. Конфигурация: Он поддерживает как Java-based, так и XML-based конфигурации, что позволяет гибко настраивать правила безопасности приложения.
7. Пользовательские расширения: Spring Security позволяет создавать собственные расширения для реализации дополнительных функций безопасности в соответствии с требованиями приложения. Это делает его адаптируемым и пригодным для использования в самых различных сценариях.

Архитектура Spring Security:



Цепочка выполнения в данном проекте:

1. Spring загружает конфигурацию из класса SecurityConfig.



Этот класс конфигурирует параметры безопасности приложения:

* *.csrf(AbstractHttpConfigurer::disable)* // Отключает защиту от CSRF.
* *.requestMatchers("/css/\*\*","/site-images/\*\*","/login","/actuator/prometheus")*

*. permitAll()* // Разрешает доступ к указанным URL без аутентификации

* *.requestMatchers("/userList", "/userUpdate/\*\*", "/userCreate", "/userDelete/").hasAuthority("admin")* // Требует роль "admin" для указанных URL.
* *.anyRequest().authenticated()* // Требует аутентификацию для всех остальных запросов.
* .*loginPage("/login")* // Настраивает страницу логина.
* *.successHandler(authHandler)* // Настраивает обработчик успешной аутентификации.
* *.logoutSuccessUrl("/"));* // Настраивает URL перенаправления после выхода.
* *return http.build();* // Возвращает конфигурированную цепочку фильтров.

2. Обработка запроса на логин:

* Пользователь отправляет запрос на логин на URL /login.
* ***UsernamePasswordAuthenticationFilter*** обрабатывает запрос, извлекает учетные данные и передает их ***AuthenticationManager***.

3. Аутентификация пользователя:

* ***AuthenticationManager*** делегирует аутентификацию ***AuthenticationProvider***.
* ***AuthenticationProvider*** использует ***CustomUserDetailsService*** для загрузки данных пользователя из базы данных по имени пользователя. Если учетные данные верны, создается объект ***Authentication***.

4. Обработка успешной аутентификации:

* ***UsernamePasswordAuthenticationFilter*** вызывает ***AuthHandler***. ***AuthHandler*** выполняет логику перенаправления после успешной аутентификации.

5. Авторизация запросов:

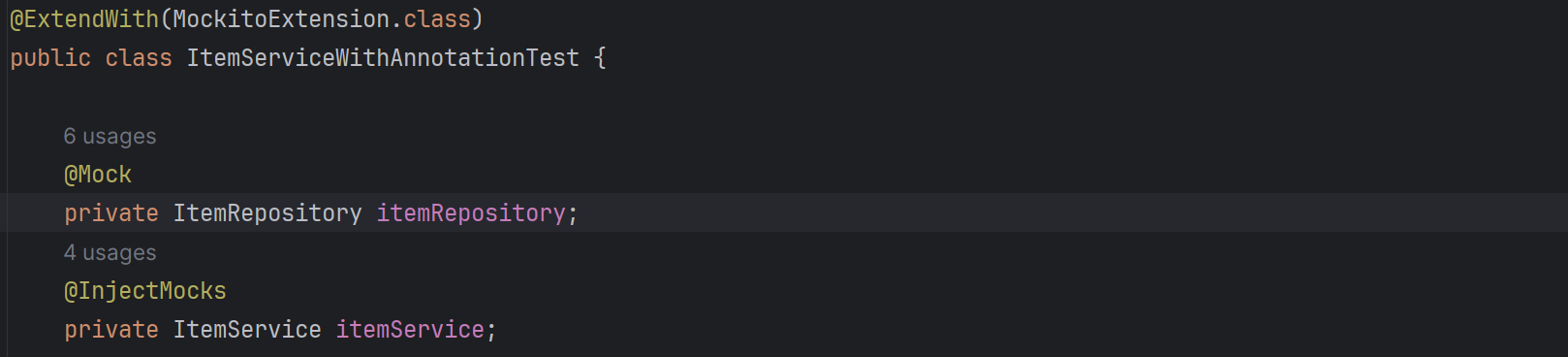
* В зависимости от URL, определённых в конфигурации, запросы авторизуются. Доступ к защищенным ресурсам предоставляется в зависимости от ролей пользователя.

**3.10 Тестирование**

В данный проект было добавлено два вида тестирования. Модульное и интеграционное.

1. Модульное тестирование нацелено на тщательную проверку каждого отдельного модуля, чтобы убедиться в его соответствии спецификациям и требованиям, а также в правильной работе в изоляции от остальных частей программы. В рамках модульных тестов проверяются входные и выходные данные, а также внутреннее состояние модуля в различных сценариях использования. Это позволяет выявлять ошибки и дефекты на ранних этапах разработки, повышать стабильность и надежность программного обеспечения. Кроме того, модульное тестирование облегчает дальнейшую поддержку и развитие программного продукта, что делает его более устойчивым и долговечным.

Пример модульного тестирования из проекта:

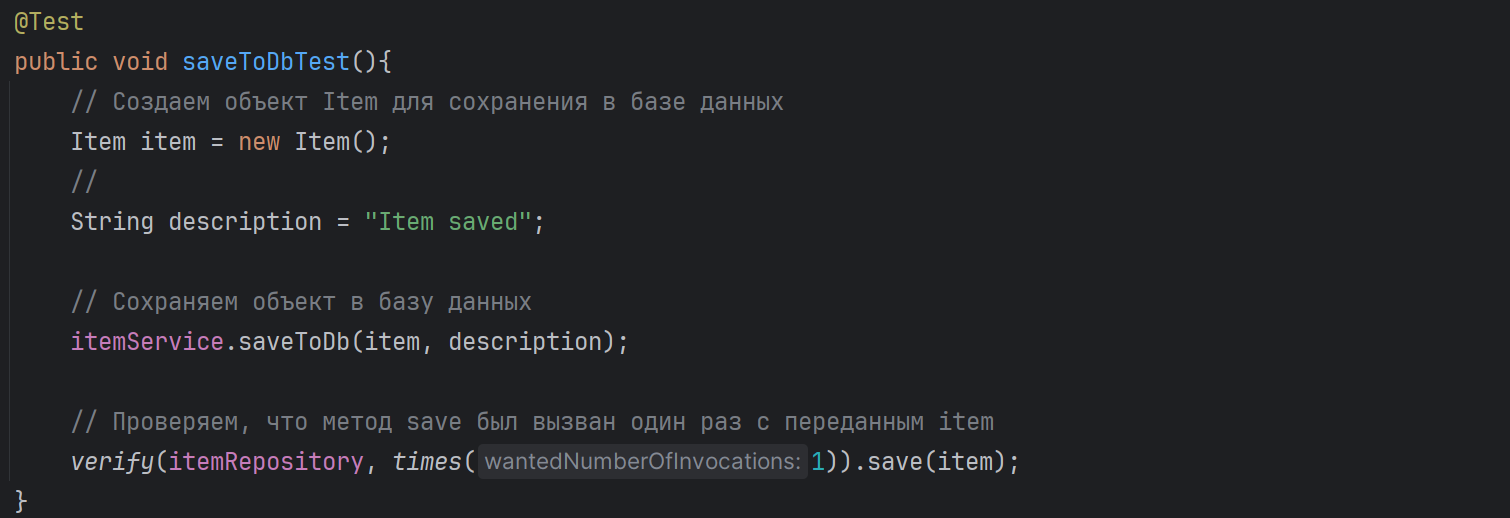


@ExtendWith(MockitoExtension.class): Эта аннотация используется в JUnit для указания расширения (extension), которое должно быть использовано при выполнении тестов. В данном случае, MockitoExtension.class - это расширение, предоставляемое библиотекой Mockito, которое обеспечивает интеграцию Mockito с JUnit.

@Mock: Эта аннотация помечает поле в тестовом классе как макет (mock). Mockito создаст объект-заглушку для этого поля, который будет использоваться вместо реального объекта в тестах. В этом примере itemRepository помечен как макет, соответственно Mockito будет создавать макет ItemRepository для использования в тестах.

@InjectMocks: Эта аннотация используется для инъекции макетов в тестируемый объект. Mockito автоматически внедрит созданные макеты (помеченные @Mock) в поля, помеченные @InjectMocks.

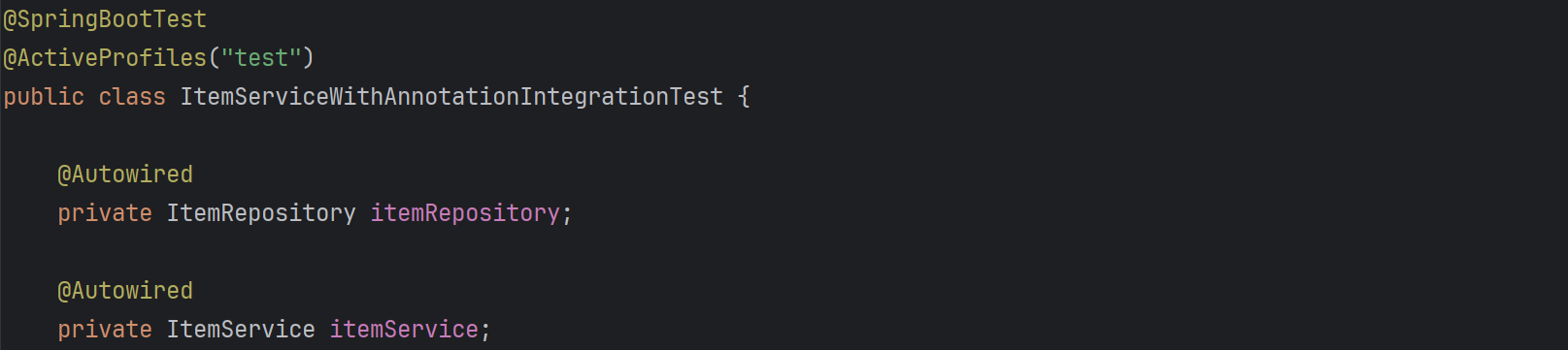
Тест saveToDbTest():



Тест проверяет корректность сохранения объекта Item в базе данных с использованием ItemService, а также проверяет, что метод save в ItemRepository вызывается правильно.

2. Целью интеграционного тестирования является обнаружение ошибок и дефектов, связанных с взаимодействием компонентов, а также уверенность в том, что система функционирует корректно в реальной среде выполнения. В интеграционных тестах проверяются такие аспекты, как передача данных между компонентами, корректность обработки сообщений и событий, а также согласованность работы системы в целом.

Пример интеграционного тестирования из проекта:



Aннотации:

@SpringBootTest: Эта аннотация указывает на то, что тестируемый класс является частью интеграционного теста Spring Boot.

@ActiveProfiles("test"): Эта аннотация позволяет установить активный профиль (profile) приложения для тестирования. В данном случае, профиль "test" активируется, что позволяет использовать специальные настройки и конфигурации для тестового окружения.

@Autowired: Эта аннотация указывает на то, что Spring должен автоматически внедрить (inject) зависимости в поле класса.

Тест daleteByIdIntegrationTest():



Этот тест проверяет корректность работы метода deleteById в классе ItemService в контексте интеграционного тестирования.

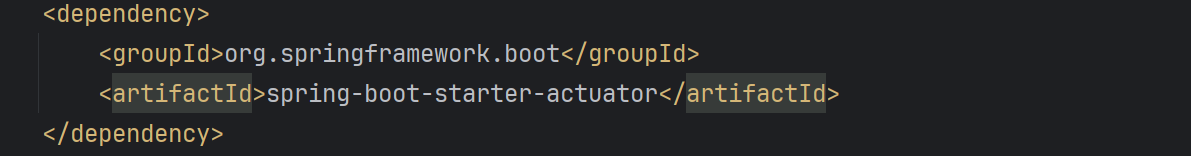
**3.11. Подключение системы мониторинга.**

Для осуществления мониторинга были использованы следующие инструменты: Actuator, Micrometr, Prometheus и Grafana.

**Настройка Actuator:**

Spring Boot Actuator - это набор инструментов для мониторинга и управления приложениями, построенных на основе Spring Boot. Он включает в себя готовые к использованию точки (endpoints). Эти конечные точки помогают администраторам и разработчикам получать информацию о состоянии приложения, его компонентов и ресурсов, а также проводить диагностику и настройку в реальном времени. Таким образом, Spring Boot Actuator значительно упрощает управление жизненным циклом приложения и способствует его эффективной эксплуатации и поддержке.

Добавляем зависимость в pom.xml:



Применяем настройки к Actuator:



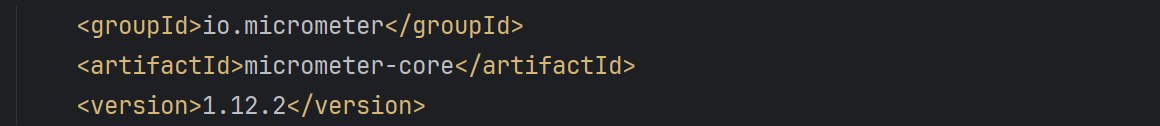
*management.endpoints.web.exposure.include: metrics, prometheus, health*:

* metrics: Позволяет включить конечную точку для сбора метрик, которая предоставляет информацию о производительности и использовании ресурсов приложения.
* prometheus: Включает конечную точку, которая предоставляет данные в формате, совместимом с системами мониторинга Prometheus.
* health: Включает конечную точку для проверки здоровья приложения, которая сообщает о состоянии различных компонентов приложения, таких как базы данных, очереди сообщений и других.

**Интеграция с Micrometer:**

Micrometer - это библиотека для мониторинга, предназначенная для приложений на базе Java. Она предоставляет единый интерфейс для сбора и экспорта метрик в различные системы мониторинга, такие как Prometheus, Grafana, Datadog, New Relic и другие. Micrometer интегрируется со Spring Boot, что позволяет легко собирать метрики приложения и системные метрики, упрощая мониторинг производительности и выявление проблем.

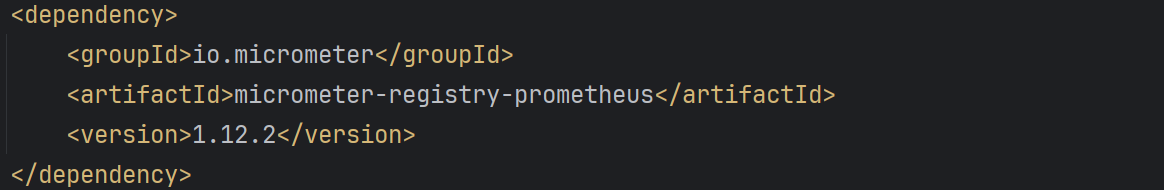
Для интеграции Micrometr в приложение необходимо подключить зависимость в pom.xml:



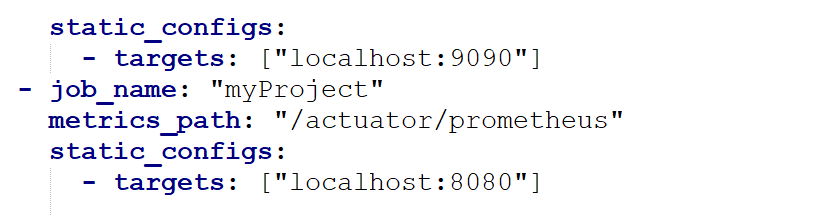
**Настройка Prometheus:**

Prometheus – это система мониторинга с открытым исходным кодом, разработанная для сбора и анализа метрик. Она отличается высокой надежностью и гибкостью, что делает ее популярным выбором для мониторинга широкого спектра систем. Prometheus широко используется для отслеживания состояния серверов, приложений, контейнеров и микросервисов. В дополнение к этому, он предоставляет мощные инструменты для визуализации и анализа данных, что упрощает диагностику и управление системой.

Для интеграции Prometheus в приложение необходимо подключить зависимость в pom.xml:

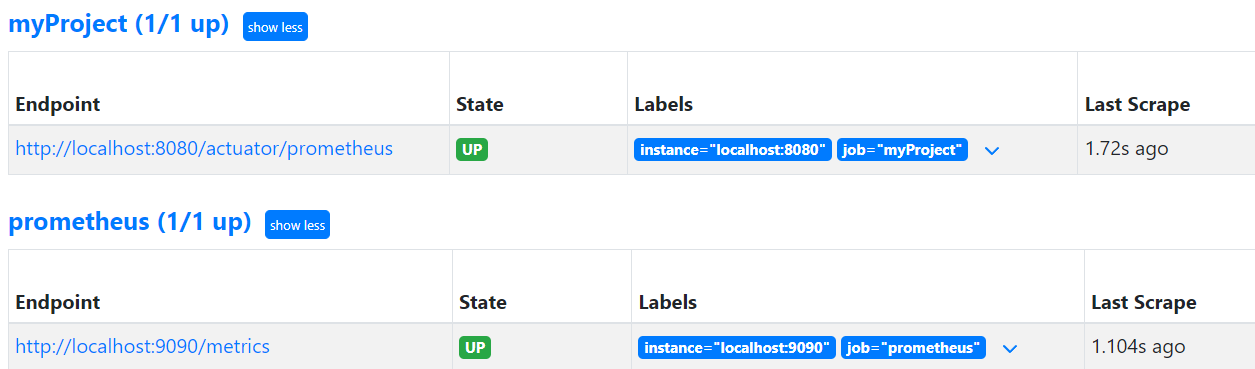


Чтобы настроить Prometheus, нужно сначала скачать и установить его для нужной операционной системы. Затем необходимо внести конфигурацию для сбора метрик от приложения в файл настроек Prometheus (prometheus.yml):



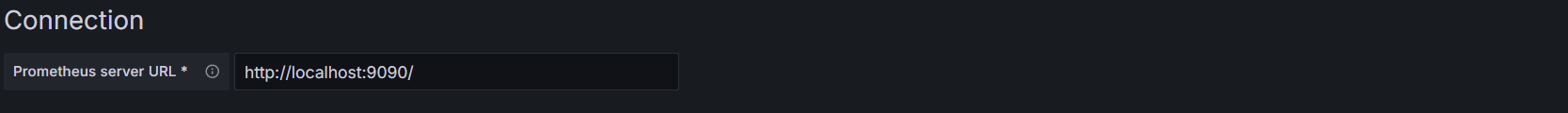
Prometheus можно запустить с помощью NSSM в качестве службы NT. NSSM позволяет запускать его как службу и перезапускает в случае сбоя.

По адресу *http://localhost:9090/* можно посмотреть подключённые эндпоинты.



**Настройка Grafana:**

Grafana – это открытая платформа для анализа и визуализации данных. Она позволяет создавать настраиваемые и интерактивные графики, панели мониторинга и оповещения на основе данных из различных источников, таких как Prometheus, InfluxDB, Elasticsearch и других. Grafana широко используется для мониторинга инфраструктуры, приложений и бизнес-процессов, предоставляя пользователям мощные инструменты для анализа данных и принятия решений. Чтобы настроить Grafana, нужно скачать и установить её для соответствующей операционной системы. Затем добавьте источник данных Prometheus в Grafana, указав URL вашего сервера Prometheus.



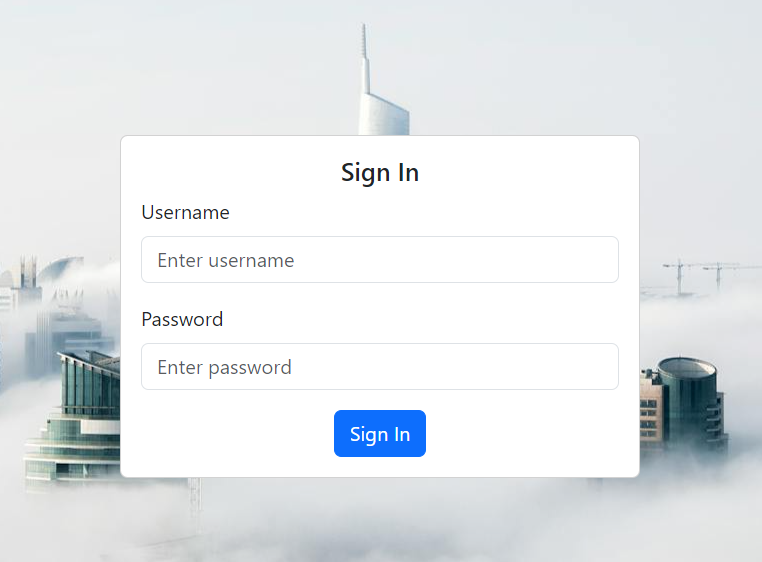
Пример Grafana dashboard:  


В совокупности Actuator, Micrometer, Prometheus и Grafana позволяют создать мощную систему мониторинга и управления для приложений на основе Spring Boot. Actuator используется для сбора информации о состоянии приложения, Micrometer для сбора и отправки метрик в Prometheus, а Grafana для визуализации этих метрик в виде красивых дашбордов. Эта комбинация инструментов обеспечивает полное представление о работе приложения и позволяет оперативно реагировать на возникающие проблемы.

**3.12 Реализация дизайна и функционала веб-приложения**

При разработке дизайна веб-приложения были использованы технологии Thymeleaf, Bootstrap и JavaScript, что обеспечило создание динамических таблиц и страниц. Эти технологии обеспечивают удобный и интуитивно понятный интерфейс для пользователей.

1. Была переопределена логин форма.



2. При успешном входе в систему открывается главная страница – main.html.



Функциональность:

* + - Пользователь может навигировать по сайту с помощью навигационной панели и иконок, переходя между различными разделами.
    - На главной странице отображается вдохновляющая цитата.

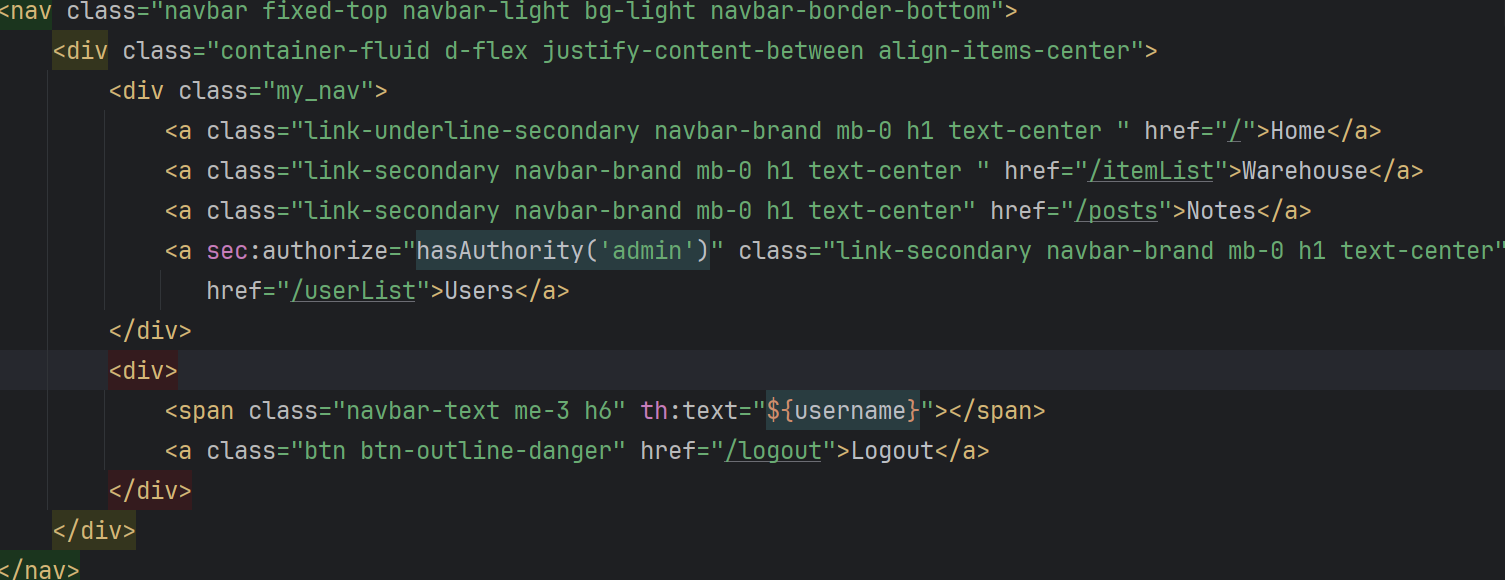
3. Навигационная панель.



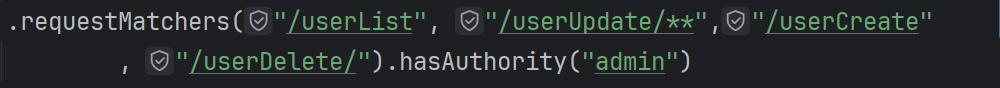
* Home: перемещает или обновляет главную страницу.
* Warehouse: перенаправляет на страницу склада.
* Notes: перенаправляет на страницу заметок.
* Users: перенаправляет на страницу управления пользователями.
* Справа отображается полное имя пользователя.
* Logout: Кнопка выхода.

Код панели навигации:

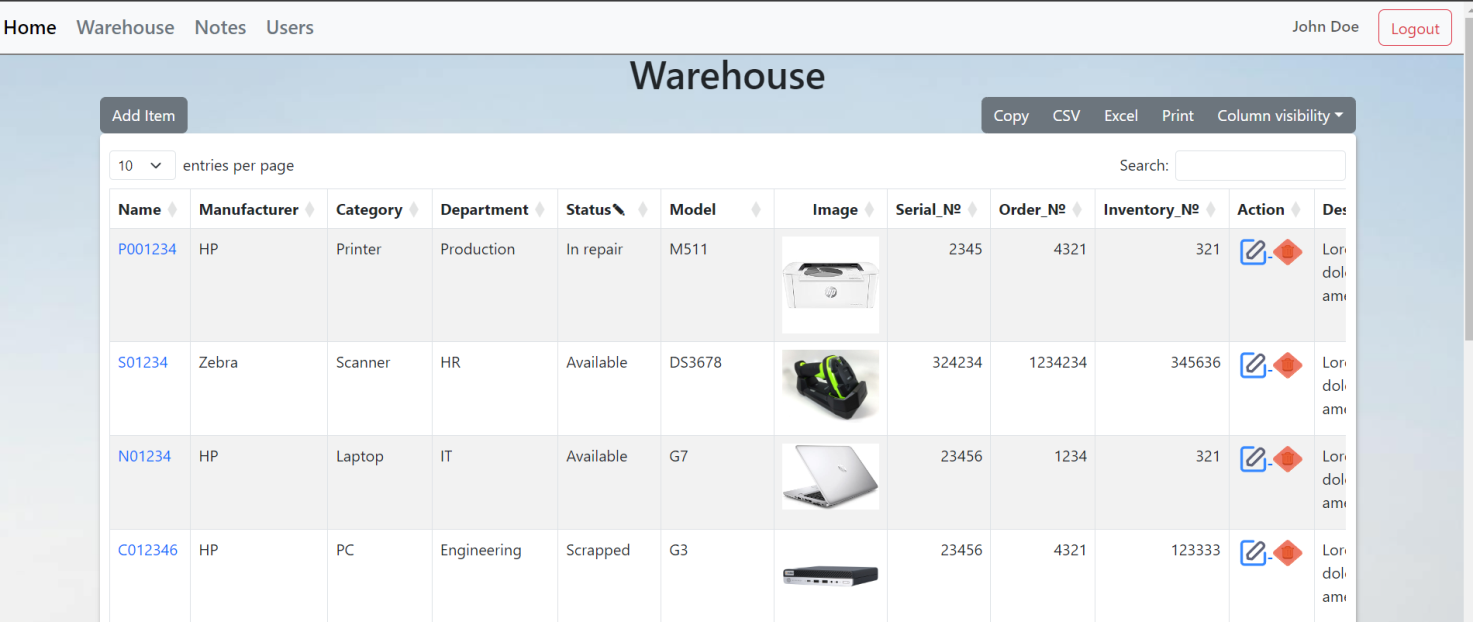
Кнопка Users отображается только пользователям с ролью admin.



Пользователи, не обладающие ролью администратора, не будут иметь доступ к странице управления пользователями. Это обеспечивается настройками в SecurityFilterChain.



4. Страница Warehouse.



Секция Warehouse:

* Отображает список товаров, представленных в виде таблицы, что позволяет удобно просматривать и управлять данными о товарах.
* Для каждого товара отображается: имя, бренд, категория, статус, модель, изображение, серийный номер, номер заказа, описание, дата добавления и дата изменения. Это обеспечивает полное представление о каждом товаре.
* Для перебора списка товаров используется цикл th:each="item : ${items}".
* Название товара (Name) является ссылкой, которая перенаправляет пользователя на страницу с подробной информацией о товаре, что упрощает доступ к детализированной информации.

Функциональность:

* Пользователь может просматривать товары, что позволяет легко получать информацию о текущем ассортименте.
* Есть возможность добавлять на склад новый товар, что помогает поддерживать актуальность данных.
* Возможность просматривать информацию о выбранном товаре упрощает управление и отслеживание товаров.
* Возможность редактирования и удаления товаров позволяет поддерживать базу данных в актуальном состоянии.
* Возможность сортировки товаров помогает быстро находить нужные позиции по различным критериям.
* Возможность поиска товаров улучшает навигацию и ускоряет процесс нахождения конкретных товаров.
* Возможность импорта в форматы CSV или XLSX облегчает интеграцию с другими системами и обработку данных.
* Возможность распечатать.

Сортировка, поиск и возможность импорта были реализованы с использование библиотеки DataTables. DataTables — это мощная библиотека JavaScript, предназначенная для улучшения HTML-таблиц. Этот гибкий инструмент, основанный на принципах прогрессивного улучшения, добавляет расширенные функции в любые HTML-таблицы. DataTables предоставляет готовые шаблоны, которые включают JavaScript, Bootstrap и CSS. Благодаря DataTables разработчики могут быстро и эффективно улучшить пользовательский интерфейс и опыт работы с таблицами.

Подключение DataTables:

* Подключение в html:

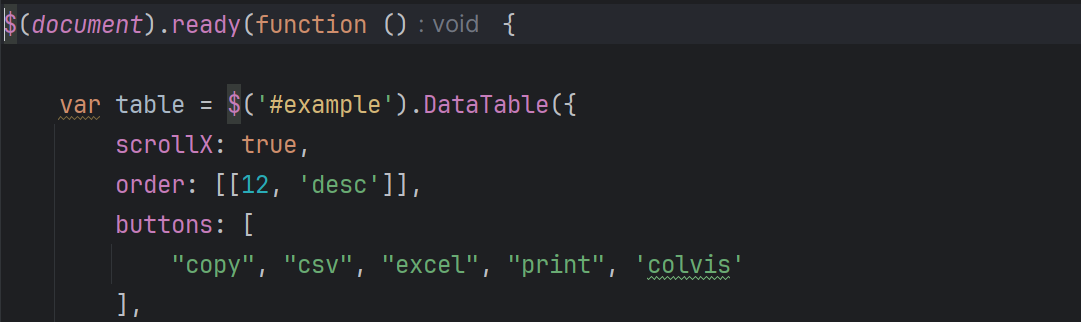
*- id=”example”*



* Импорт CSS и JavaScript:

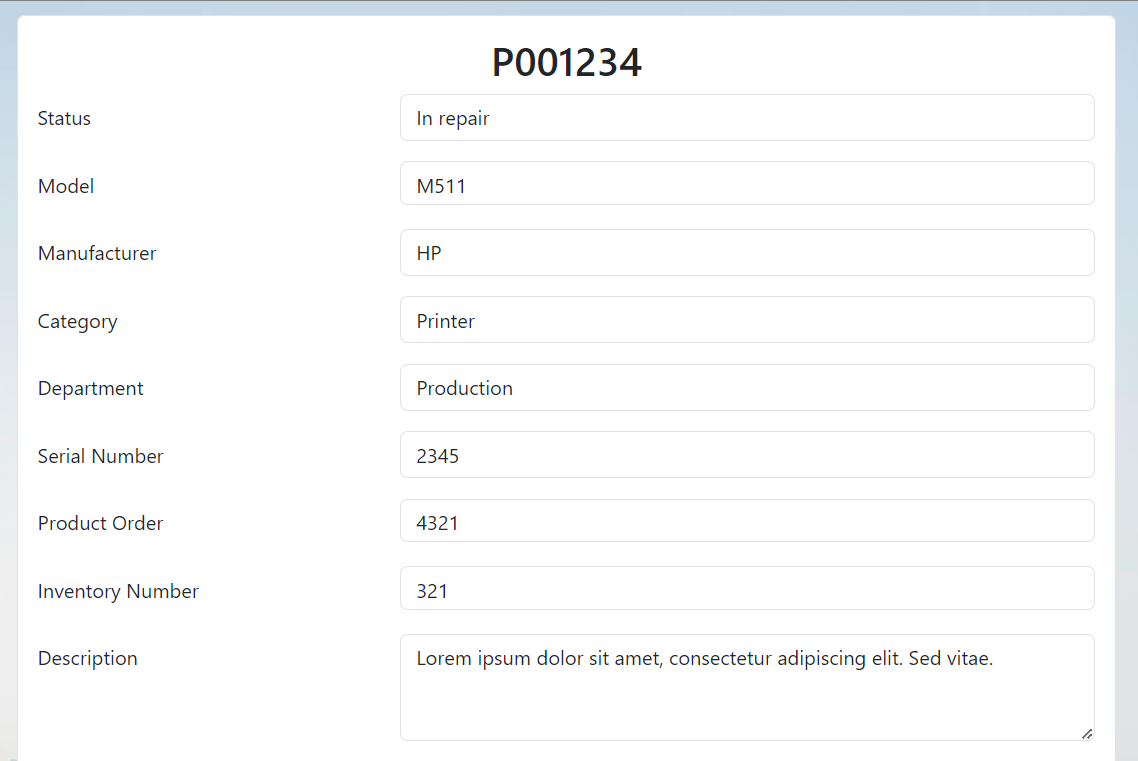


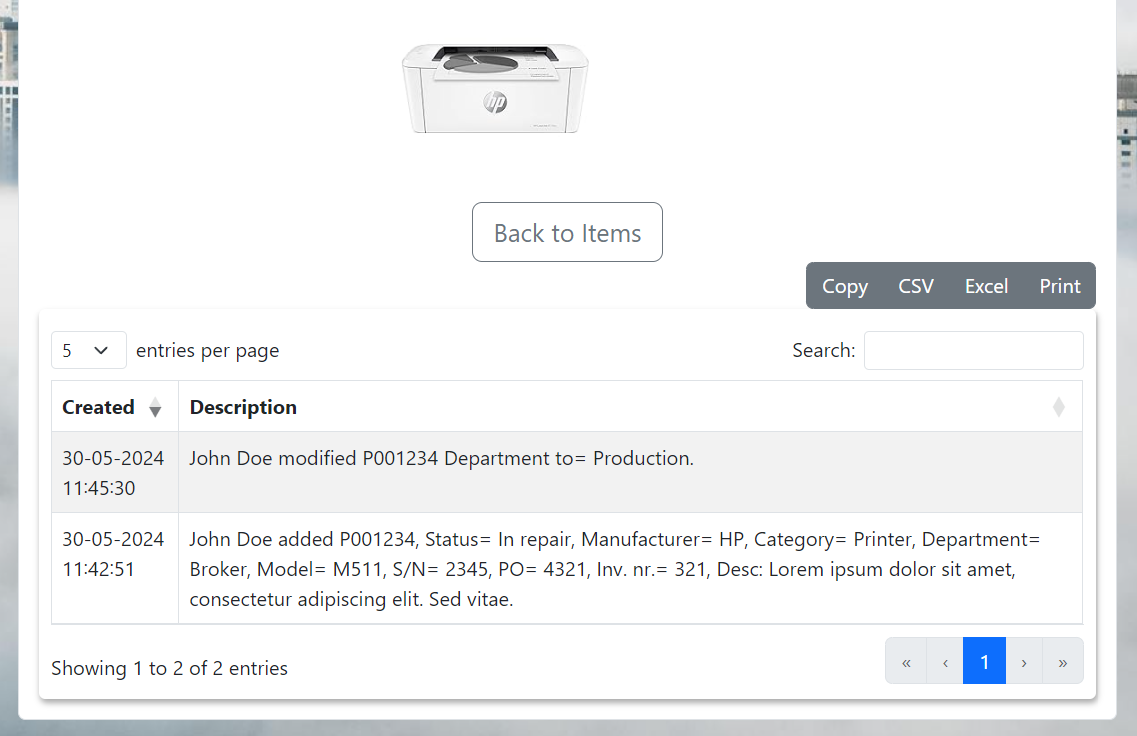
* Подключение в myScript.js



Этот скрипт инициализирует таблицу с данными на веб-странице. Таблица имеет горизонтальную прокрутку и сортировку по 12-му столбцу в порядке убывания. Также добавлены кнопки для копирования, экспорта в CSV и Excel, печати и управления видимостью столбцов. Эти настройки позволяют пользователю легко работать с большими объемами данных и экспортировать их в разные форматы.

5. Информация о товаре.

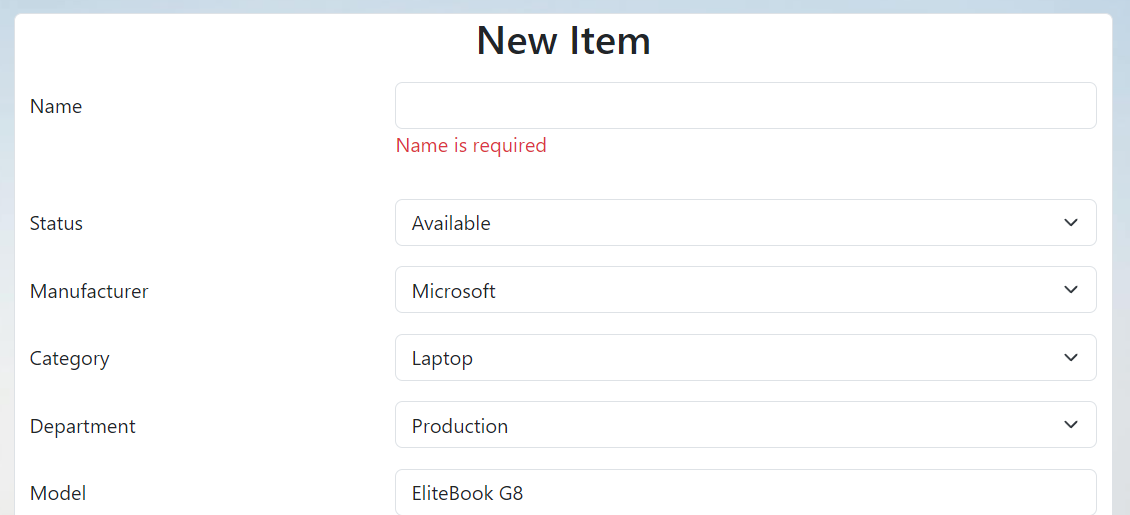


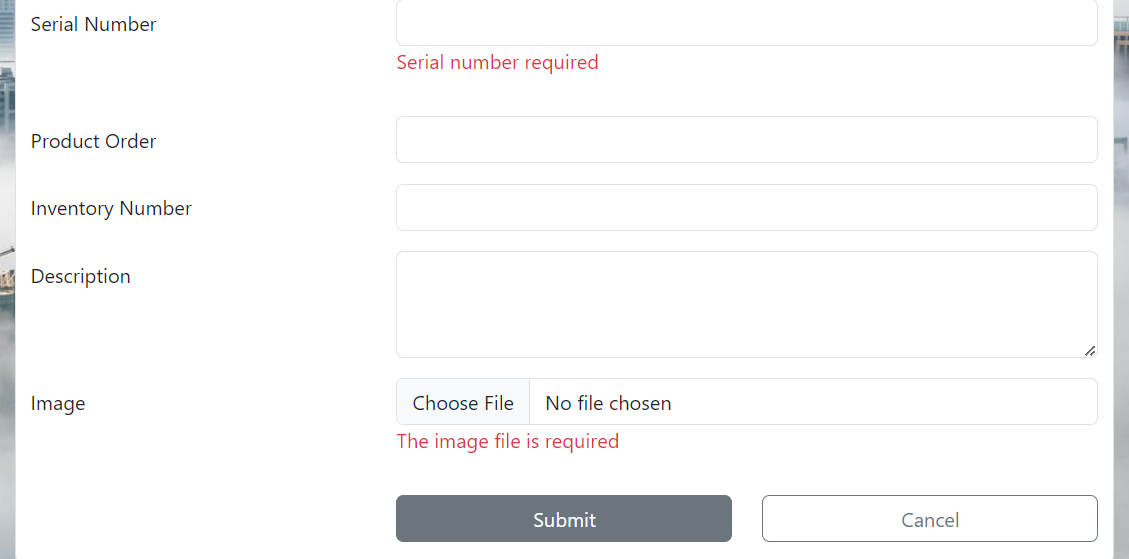


Функциональность:

* Просмотр подробной информации о товаре.
* Возврат на страницу с товарами.
* Просмотр подробного описания логов.
* Поиск логов.
* Импорт логов в форматы CSV или XLSX.
* Печать логов.

6. Добавление товара.



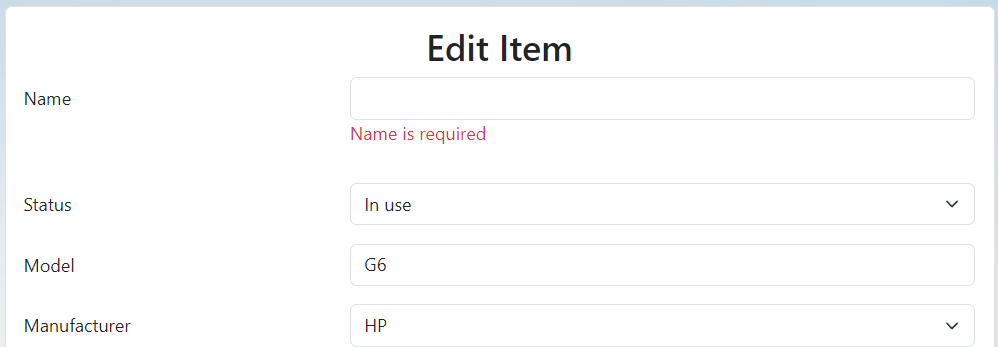


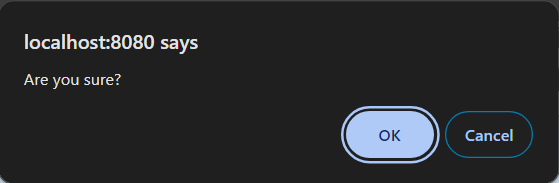
Функциональность:

* Добавление нового товара на склад.
* Проверка полей на ошибки с помощью объекта ItemDto.
* Возврат к списку товаров при нажатии кнопки "Cancel".

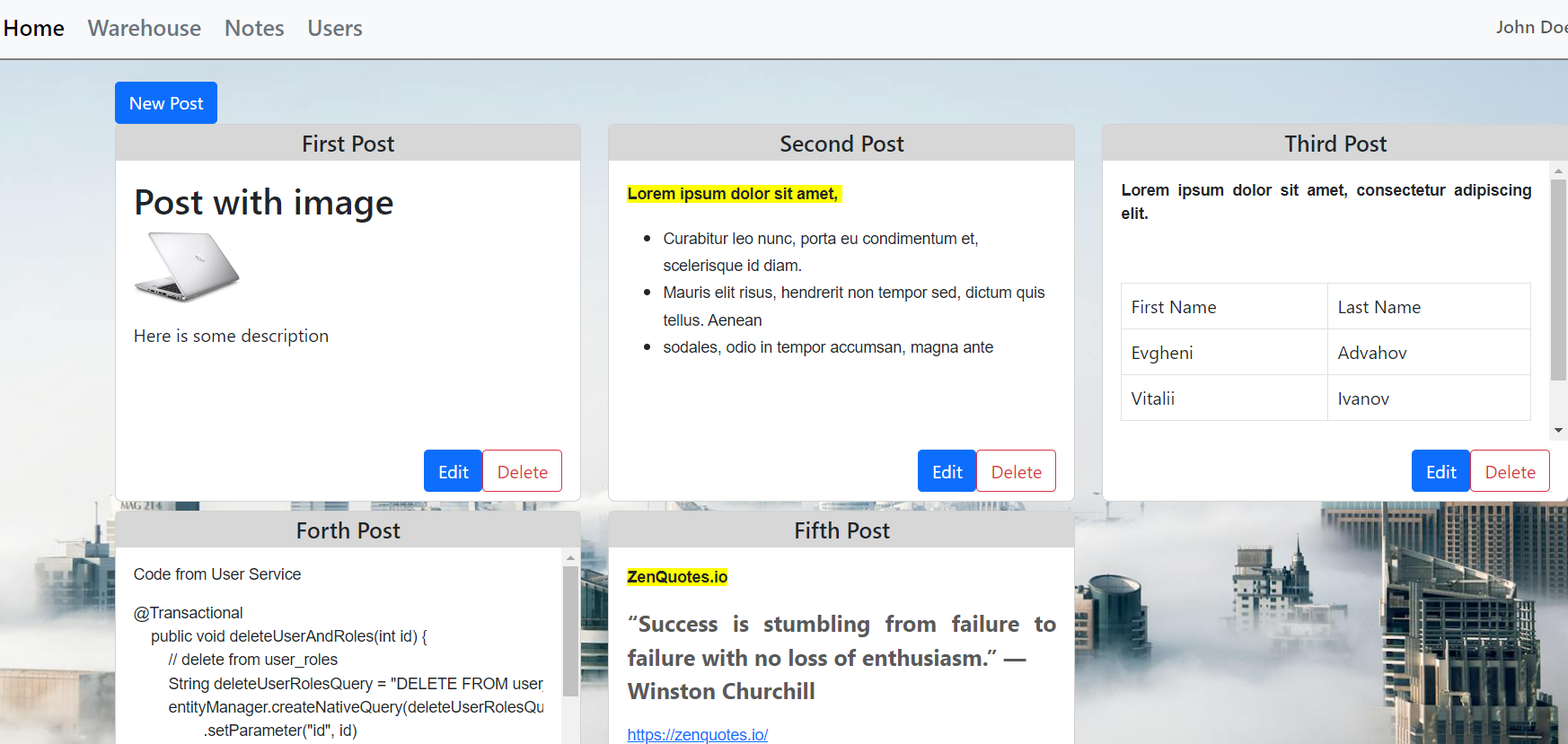
7. Редактирование товара.

Редактирование товара происходит аналогично добавлению нового товара. Проверка полей на ошибки также осуществляется.



8. Удаление товара.  


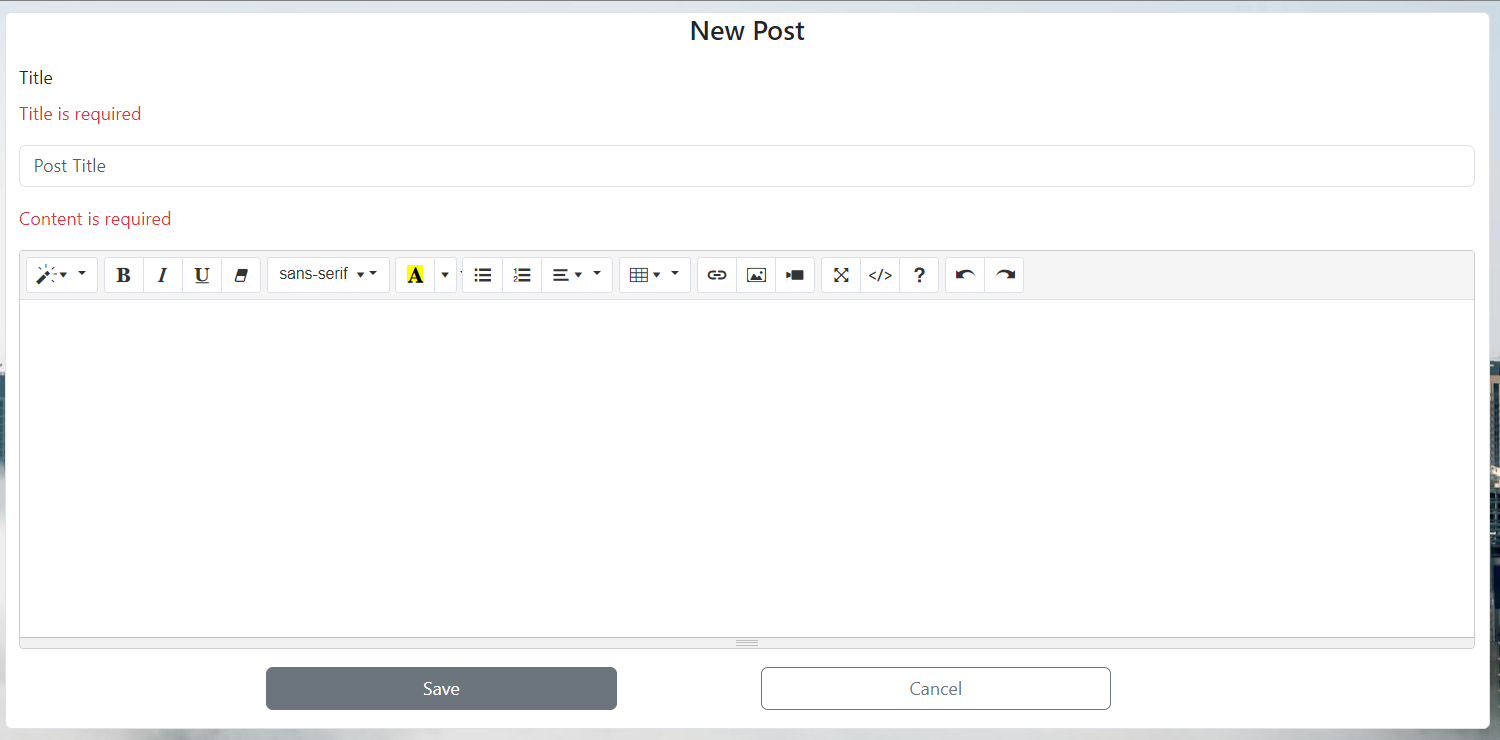
9. Страница Notes.



Функционал:

* Отображение заметок в виде карточек.
* Возможность создать, отредактировать или удалить заметку.

10. Создание новой заметки.



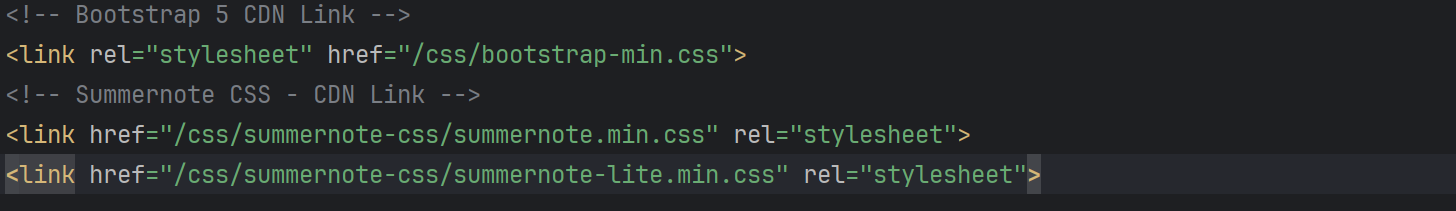
Функционал:

* Подключён Summernote.
* Обрабатываются ошибки с помощью класса PostDto.

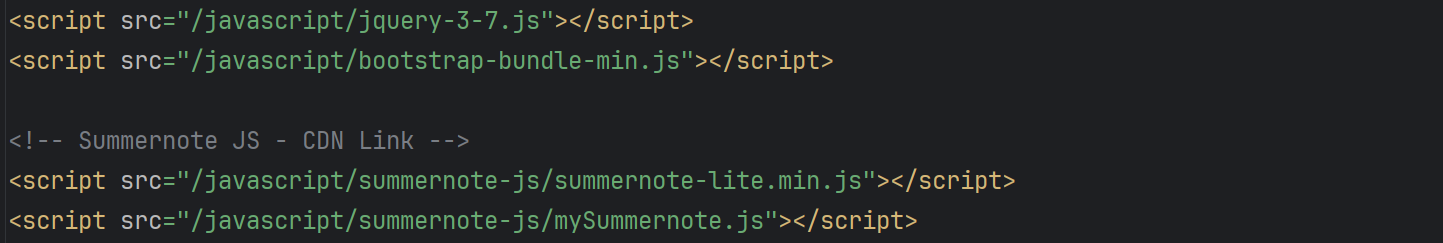
Summernote - это простой и удобный в использовании WYSIWYG (What You See Is What You Get) текстовый редактор для веб-приложений. Он позволяет пользователям форматировать текст, добавлять изображения, таблицы, ссылки и другой контент, подобно тому, как это делается в текстовых процессорах.

Подключение Summernote.

* Подключаем стили:



* Подключаем JavaScript:



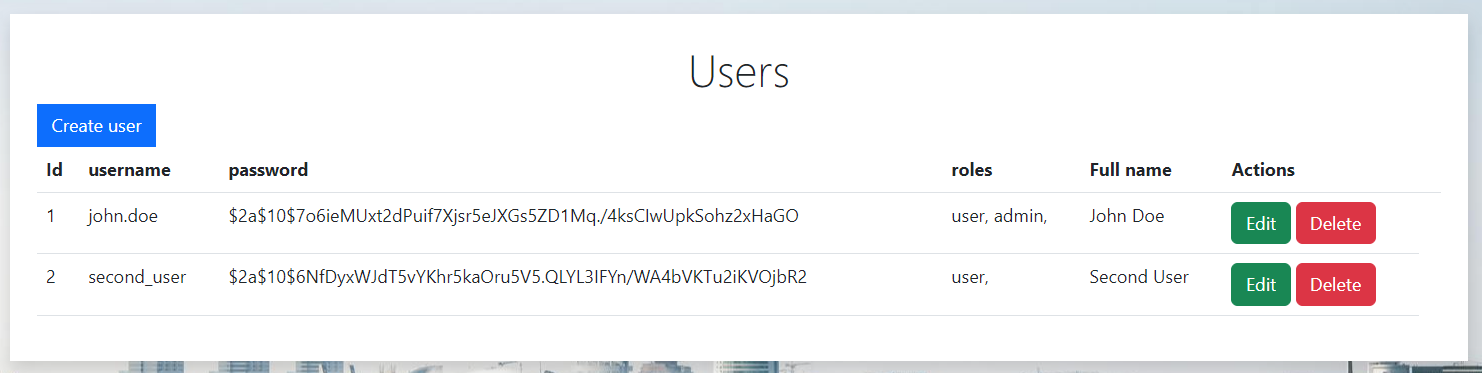
* Создаём текстовое поле в createPost.html:



* Инициализируем Summernote в mySummernote.js:



11. Страница Users.



Функционал:

* Возможность создать, отредактировать или удалить пользователя.
* Пароли хранятся и отображаются в зашифрованном виде.

**Заключение**

В ходе дипломного проекта было создано веб-приложение для управления складом и заметками с использованием Java Spring Framework, Bootstrap и MySQL.

Реализованы:

1. Интуитивный интерфейс.
2. Функционал добавления, редактирования и удаления товаров и заметок.
3. Обеспечена безопасность с помощью Spring Security.
4. Применены принципы SOLID и паттерны проектирования.

Дальнейшее развитие включает:

1. Улучшение интерфейса.
2. Расширение функционала сортировки и фильтрации.
3. Доработку механизмов безопасности.

4. Интеграцию с дополнительными сервисами или API для расширения возможностей приложения.

5. Внедрение инструментов для аналитики, отчетности и статистики.

Данный проект является демонстрацией навыков разработки веб-приложений с использованием современных технологий и подтверждением умения применять их на практике.

Ссылка на проект: <https://github.com/EvgheniAdvahov/myProject>

**Список источников**

1. Учебный материал GeekBrains.

2. Статьи и видеоматериалы из сети интернет.

3. ChatGPT.